SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

AD-A241 225

1b. RESTRICTIVE MARKINGS	
3. DISTRIBUTION/AVAILABILITY OF REPORT	Approved for public release; distribution is unlimited.
4. PERFORMING ORGANIZATION REPORT NUMBER(S) MIT/LCS/TR 514	5. MONITORING ORGANIZATION REPORT NUMBER(S) N00014-89-J-1988
6a. NAME OF PERFORMING ORGANIZATION MIT Lab for Computer Science	6b. OFFICE SYMBOL (If applicable)
7a. NAME OF MONITORING ORGANIZATION	Office of Naval Research/Dept. of Navy
6c. ADDRESS (City, State, and ZIP Code) 545 Technology Square Cambridge, MA 02139	7b. ADDRESS (City, State, and ZIP Code) Information Systems Program Arlington, VA 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION DARPA/DOD	8b. OFFICE SYMBOL (If applicable)
9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
8c. ADDRESS (City, State, and ZIP Code) 1400 Wilson Blvd. Arlington, VA 22217	10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO	PROJECT NO.	TASK NO	WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)

Algorithms for Scheduling and Network Problems

12. PERSONAL AUTHOR(S)

Joel Martin Wein

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM ______ TO ______	August 1991	135

16. SUPPLEMENTARY NOTATION

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)

Scheduling, combinatorial optimization, network algorithms, parallel computation, connection machine, approximation algorithms, online algorithms.

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

In this thesis we develop algorithms for two basic classes of problems in combinatorial optimization: *deterministic machine scheduling* and *network optimization*.

In the first part of the thesis we consider approximation algorithms for two basic scheduling environments: *shop scheduling* and *parallel machine scheduling*. We give approximation algorithms for shop scheduling that significantly improve upon the performance of previous algorithms. We then study *on-line* approximation algorithms for parallel machine scheduling.

In the second part of the thesis we present several theoretical and practical results about parallel algorithms for network optimization problems, including

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Carol Nicolora	22b. TELEPHONE (Include Area Code) (617) 253-5894 / 22c. OFFICE SYMBOL

19 a.

- A *Las Vegas* $\mathcal{RNC}$ algorithm for the minimum-weight perfect matching problem when the weights are input in unary.
- A proof that *approximating* the value of the minimum-cost maximum flow is $\mathcal{P}$-Complete.
- An experimental study of implementations of algorithms to solve the dense assignment problem on a real massively parallel computer, the Connection Machine CM-2.

Algorithms for Scheduling and Network Problems

by

Joel Martin Wein

A.B., Applied Mathematics
Harvard University
(1985)

Submitted to the Department of Mathematics
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1991

Signature of Author______________________________

Department of Mathematics
August 9, 1991

Certified by______________________________

David B. Shmoys
Assistant Professor of Industrial Engineering and Operations Research,
Cornell University
Thesis Supervisor

Accepted by______________________________

Alar Toomre
Chairman, Applied Math Committee

Accepted by______________________________

Sigurdur Helgason

Algorithms for Scheduling and Network Problems

by

Joel Martin Wein

Submitted to the Department of Mathematics

on August 9, 1991,

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Abstract

In this thesis we consider the development of algorithms for two very basic classes of problems in combinatorial optimization: *deterministic machine scheduling* and *network optimization.* Most of the scheduling problems we consider are $\mathcal{NP}$-Complete and therefore we study *approximation algorithms* for these problems. The network problems are all known to be polynomial time-solvable; we will be interested in *parallel algorithms* for these problems. An important connection between the two topics of this thesis is that the scheduling problems we consider are related to problems that arise in the design of parallel computers; they are also, however, basic building blocks in the theory of combinatorial scheduling.

We consider the two basic environments for scheduling a set of machines: *shop scheduling* and *parallel machine scheduling.* In the *shop scheduling* problem we are given m machines and n jobs; a job consists of an ordered set of operations, each of which must be processed on a specified machine. The aim is to complete all jobs as quickly as possible. This problem is strongly $\mathcal{NP}$-hard even for very restrictive special cases, and very little was known about approximation algorithms for it. We give the first randomized and deterministic polynomial-time approximation algorithms that yield polylogarithmic approximations to the optimal length schedule. We also give the first *parallel* approximation algorithms for shop scheduling. Most of our results apply to the important generalization in which there are m' types of machines, a specified number of machines of each type, and each operation must be processed on one of the machines of a specified type.

In the *parallel machine* environment a job consists of *one* operation which can be processed on any of the machines. Most of the basic questions about approximation algorithms for these problems have already been answered, but the algorithms that have been developed are typically *off-line*, meaning that they must be given the entire specification of a problem instance before they begin to construct a schedule. This does not model many real-world problems, including scheduling of jobs on a multiprocessor. We study the problem of scheduling jobs on parallel machines in an *on-line* fashion, where the existence of a job is not known until an unknown *release date*, and the processing requirement of a job is not known until the job is processed to completion. Despite this lack of knowledge of the future, we wish to schedule so as to minimize the completion time of the entire set of jobs.

We give two rather general techniques to convert algorithms that require more knowledge about the input data into algorithms that need less advance knowledge. As a result we are able to give on-line approximation algorithms for all of the fundamental parallel machine models. In most of these models we are able to show that our algorithms are asymptotically optimal.

In the second part of this thesis we consider both theoretical and practical issues in the design of parallel algorithms for network optimization problems. We first consider the minimum weight perfect matching problem, where the weights are input in unary. All previous $\mathcal{RNC}$ algorithms for this problem were *Monte-Carlo* algorithms, which produced a correct solution with high probability but gave no indication when they failed. We give a *Las Vegas* $\mathcal{RNC}$ algorithm for the problem – one that certifies, with high probability, that its solution is correct – utilizing reductions between minimum weight perfect matching and the T-join problem. We also show how to apply the technique to a number of other combinatorial problems.

We then consider the problem of finding a minimum-cost maximum flow in a network. This problem is known to be $\mathcal{P}$-Complete, and therefore it is expected that there exists no $\mathcal{NC}$ algorithm for this problem. We prove that even *approximating* the value of the minimum-cost maximum flow is $\mathcal{P}$-Complete.

Finally we present an experimental study of implementations of algorithms to solve the dense assignment problem on a real massively parallel computer, the Connection Machine CM-2. We describe the implementations of five different algorithms for the problem, including a new hybrid approach that we developed, and discuss other approaches which we did not implement. We evaluate the implementations empirically; the best proves to be the hybrid auction algorithm.

Keywords: Scheduling, Combinatorial Optimization, Network Algorithms, Parallel Computation, Connection Machine, Approximation Algorithms, On-line Algorithms.

Thesis Supervisor: David B. Shmoys

Title: Assistant Professor of Industrial Engineering and Operations Research, Cornell University

Acknowledgments

I am very lucky to have had David Shmoys as my thesis advisor. He has been extraordinarily giving of his time, ideas, advice, and encouragement during all phases of my career at MIT. He has been patient and supportive when things were going slowly, and enthusiastic when things were going well. He has been accessible at practically every time of day or night, even immediately after the birth of a daughter. Despite the fact that he has been at Cornell the past two years, the responsible way in which he has continued to carry out his role as my advisor has enabled me to finish this thesis. He is a model for me of a teacher, advisor and researcher. He has also been a valuable travel consultant, political commentator and a good friend. Chapters 2 and 3 are joint work with him.

I am also lucky to have learned much of what I know about combinatorial optimization in courses taught by Éva Tardos. I have gained much from discussions with her; in particular I thank her for suggesting that I consider T-joins in the work presented in Chapter 6, and for pointing out the planar multicommodity flow application as well.

The wonderful courses on parallel computation I took from Tom Leighton and Charles Leiserson during my first year of graduate school were a major component in my decision to pursue research in theoretical computer science. They have both provided me with much good advice, technical and otherwise, during the past five years. I am particularly grateful to Tom for helping to "take care of me" after David moved to Cornell, and to Charles for putting me in contact with Thinking Machines Corporation. I also thank them for serving on my thesis committee. I have learned a tremendous amount from Baruch Awerbuch while at MIT, and I thank him for serving on my thesis committee as well.

My fellow students Cliff Stein and David Williamson have both been good friends and valued collaborators during my career at MIT. They have listened to a remarkable number of my practice talks and read a remarkable number of drafts of my papers. They have always been ready to listen to crazy ideas on any topic, and to provide many of their own. I will especially miss discussions with Cliff on professional basketball and with David on theology. Chapters 2 and 5 are joint work with Cliff; Chapter 3 is joint work with David.

I can't imagine a more stimulating and friendly research environment than the theory of computation group at MIT. The special character of the group is due to the efforts of many people, and I am grateful to all of them for the numerous ways in which they have enriched my life here at MIT. I am similarly grateful to my friends and colleagues in the math department. Special thanks to the award-winning Be Hubbard and Phyllis Ruby for all the things they do so well, and to David Jones and David Wald for infinite amounts of help with LaTeX.

During several summers and vacations I have had the wonderful experience of working in the MCSG group of Thinking Machines Corporation. Much thanks to Jill Mesirov for her support and advice, and for providing this opportunity. While there I met Stavros Zenios, whose energy, enthusiasm and good ideas have been a pleasure to experience. I thank him for the many ways in which he has helped me. Chapter 7 is joint work with him that was largely carried out while I was at Thinking Machines. I also thank the members of MCSG who helped make that work possible.

I acknowledge the support of an Army Research Office graduate fellowship, Air Force Contracts AFOSR-86-0078 and AFOSR-89-0271, DARPA contract N00014-89-J-1988, the National Science Foundation under a PYI awarded to David Shmoys, with matching support from IBM, SUN and UPS, and the Cornell Computational Optimization Project.

Much gratitude to Oliver Karlin and Terry Sanger, who have been wonderful friends over the last ten years. The same to all my friends from Harvard Hillel, who have helped me grow in ways more important than as a scientist.

Most importantly, I wish to thank my family, that which I started with and that which I acquired while at MIT. My in-laws, Michael and Elaine Abend, Gina and Jeff Emdin, and Leone Drobes have given so much of themselves to me the last few years that I already know I will never be able to repay them. My sister Debra has been a terrific sibling without whose love, understanding and encouragement I never would have made it this far, and certainly would not have enjoyed it as much. My parents, Bertram and Betty Wein, have always had huge amounts of confidence in me, and have made many sacrifices over the years so that I would get the best possible education. It is their love of learning and inquiry that is probably the biggest factor in the life direction I have chosen. Their love and support at a million points along the path (remember my sixth grade music course, Mom? science contest posters, Dad?) made it all possible.

The most important event of the last five years was my marriage to my wonderful wife Marjorie. She has shared in all the successes and setbacks of my research, patiently endured many late nights of work, and has provided love, support and inspiration. I have learned a great deal from her, and I hope that I can be as supportive of her when she finishes her thesis as she has been of me. It is because of all the good things I have received from my entire family that I have been able to complete this work and I dedicate it to them.

Finally, I express my gratitude to God for giving me the privilege of spending the last five years learning about his world, and contributing just a little bit more to our knowledge of it.

Contents

Introduction

In many areas of human endeavor the problem arises of choosing the "best" of a large set of possibilities. For example, an employer may want to choose a schedule for her employees that maximizes their productivity, a trucking company may want to find the least expensive way to transport goods across the country, or a city planner may want to find the optimal places in which to situate firehouses in order to maximize the safety of the city. These are all *optimization* problems: problems which require the optimization of a function of some set of variables, subject to certain constraints on the values of the variables. *Combinatorial optimization* refers to the class of optimization problems that require the choice of one solution from a finite set of discrete solutions, where the constraints and the variables typically describe the combinatorial structure of the problem.

There is a rich and longstanding relationship between combinatorial optimization and computer science. The development of efficient algorithms to solve basic problems in combinatorial optimization has been a major component in the growth of the theory of algorithms. In addition, a number of the resource allocation problems that arise in the design and control of computer systems can be modeled as problems in combinatorial optimization.

The introduction and growing popularity over the last decade of *parallel* computers has motivated a large and important group of problems in both of these categories. There are a variety of large optimization problems that would be desirable to solve in real time. One can imagine using optimization techniques to direct traffic in a congested city during rush hour, or to coordinate a large fleet of service people to respond to requests, or to dynamically schedule airplanes so as to avoid collisions. It is therefore important to understand on what sorts of optimization problems parallel computing will yield significant speedups and on which it will not.

Furthermore, the design and control questions about parallel computing systems are quite different than those about sequential machines, since they require the allocation of tasks and resources to a large number of separate processors or processes, in contrast to just one or a few. Therefore, questions about the scheduling of or the allocation of resources to multiple machines take on a much greater importance.

In this thesis we will explore both aspects of the relationship between parallel computation and combinatorial optimization by studying basic algorithmic questions in two fundamental areas of combinatorial optimization: scheduling theory and network optimization. In the first section we will study approximation algorithms for the two classic models of scheduling a set of machines: the *parallel machine* model and the *shop* model. These are in and of themselves basic problems in combinatorial optimization, but are particularly appealing to us because they bear some relationship to scheduling questions about parallel computers. The former is closely related to parallel computation since it models the scheduling of tasks on a multiprocessor [8, 93], whereas the latter has found applications to packet routing in parallel and distributed networks[25, 82, 88].

Before our work relatively little was known about approximation algorithms for shop scheduling. In Chapter 2 we give approximation algorithms for several variants of this problem that achieve significantly better performance guarantees than previous algorithms. We also give the best *parallel* approximation algorithms for several of these problems.

In contrast to shop scheduling, the state of the art in approximation algorithms for parallel machine scheduling prior to our work was excellent. Almost all of the known approximation algorithms, however, were *off-line* algorithms, in that they required the entire specification of the problem in advance. This situation does not reflect that of many real world scheduling problems, since often the scheduler does not have in advance complete knowledge of a job's running time, or of what jobs will be created and require processing in the future. In Chapter 3 we study *on-line* algorithms for scheduling parallel machines, giving matching or near-matching upper and lower bounds on how well an on-line scheduler can perform in each of the fundamental models of parallel machine scheduling.

In the second section of this thesis we turn to the design and analysis of parallel algorithms for problems in network optimization. We focus on two basic problems: that of finding a maximum weight matching in a graph, and that of finding a minimum cost flow in a network.

Both the minimum-cost flow problem and the simpler maximum-flow problem are $\mathcal{P}$-Complete problems, and therefore it is widely believed that there are no "fast" parallel algorithms to solve them. By "fast" we mean algorithms that, in a PRAM model of computation with a number of processors polynomial in the size of the input, solve the problem in worst case time polylogarithmic in the size of the problem. In chapter 5 we give an interesting separation between the parallel complexity of these two problems, showing that the minimum-cost maximum flow problem can not be approximated in $\mathcal{NC}$ unless $\mathcal{P} = \mathcal{NC}$. In contrast, it is known that the maximum flow problem can be approximated arbitrarily closely in $\mathcal{RNC}$.

In Chapters 6 and 7 we consider both theoretical and practical issues in the parallel solution of weighted matching problems. Theoretically "fast" randomized parallel algorithms for the minimum weight perfect matching problem, when the weights are input in unary, were given by Karp, Upfal and Wigderson and by Mulmuley, Vazirani and Vazirani. These algorithms produced a correct solution with high probability, but could not distinguish between success and failure. In Chapter 6 we give a *Las Vegas* algorithm for the problem, one that with high probability produces a correct solution and otherwise indicates it has failed. The technique is fairly general and has applications to a number of other combinatorial problems.

The assignment problem is the special case of the minimum-weight perfect matching problem when the graph is bipartite. In chapter 7 we study the design and implementation of algorithms to solve this problem on the Connection Machine CM-2, a massively parallel computer. We describe the implementations of five different algorithms and evaluate their performance experimentally. The best proved to be a new hybrid approach which we developed that is able to take advantage of two different levels of the parallelism of the Connection Machine. We also discuss and attempt to evaluate the other approaches which we did not implement.

Chapter 1 of this thesis gives an introduction to scheduling theory, the scheduling models we will be considering, and our results, which are presented in the following two chapters. Similarly Chapter 4 gives an introduction to parallel network optimization and our results, which are presented in Chapters 5, 6 and 7.

Chapter 1

Scheduling Theory: An Introduction

Scheduling theory is concerned with the *optimal allocation of scarce resources to activities over time* [81]. The subject has been of interest to the human race since the first time two human beings wished to use the same resource and chose to settle the matter without bloodshed. More recently, the industrial revolution and the invention of the computer have generated a huge assortment of scheduling problems, arising in areas such as manufacturing, production planning and computer control.

The most studied area in this discipline has been *deterministic machine scheduling.* By *deterministic* we mean that all of the information that defines a problem instance is determined with certainty in advance. By *machine scheduling* we mean that the resource to be scheduled is either one or a set of machines, where a machine can perform at most one activity at any time. There are a staggering number of different problems in this field: an expert system for the classification of these problems recognizes 4,536 different scheduling problems, of which 3,817 have been proved $\mathcal{NP}$-hard, 416 are known to be solvable in polynomial time, and 303 are still open [78].

Despite its size, this huge collection of problems can be neatly organized according to the following three criteria: the *machine environment*, the *job characteristics* and the *optimality criterion.* An instance of a problem is specified by a machine environment, a set of jobs that may have some specified characteristics, and an optimality criterion. Each job in the set has specific processing requirements on one or several of the machines; the goal is to produce a schedule of jobs on machines that achieves the optimal value of the optimality criterion.

The different types of machine environments can be classified into three basic categories: the *single machine* environment, the *parallel machine environment* and the *shop environment.* In this thesis we will focus on the latter two environments. This focus is to a large extent motivated by the connections between these environments and parallel processing. We will also focus almost exclusively on the optimality criterion of "completion time"; i.e. the goal of the scheduler will be to produce the "shortest" schedule – a schedule that minimizes the completion time of the last job to complete. Other typical criteria that have been considered in the literature are the *sum* of the completion times of the jobs, the *weighted sum* of the completion times of the jobs, and the *maximum lateness* of a job, where each job has an associated due date.

Approximation Algorithms: Most of the scheduling problems we will be considering are $\mathcal{NP}$-Complete, and therefore we will focus on obtaining polynomial-time algorithms for these problems that give good approximations to the optimal solution. We will evaluate an *approximation algorithm* in terms of its performance guarantee, or in other words, its worst-case relative error. Let $C^*_{\max}$ be the maximum-completion time of a job in the optimal solution. If a polynomial-time algorithm always delivers a solution of maximum-completion time at most $\rho C^*_{\max}$, then we shall call it a ρ-approximation algorithm.

1.1 The Shop Environment

Introduction

Shop scheduling refers to a large class of problems that typically arise in a shop, factory or assembly line setting. The shop has m machines, and in the basic environment each machine is different and performs a different function. Each job consists of a set of *operations*, each of which must be processed on a particular machine; a job may have more than one operation on a particular machine. We wish to produce a *schedule* which assigns a period of time to each operation during which it is processed on the appropriate machine. The goal is to minimize the *completion time* of the last operation to complete, while ensuring that no more than one operation is assigned to a machine at any point in time and no two operations of the same job are scheduled simultaneously.

A variety of constraints may be introduced on the order of execution of the operations of the job, and different sorts of constraints yield different well-known versions of the problem. (Note that we only focus on order constraints between the operations of each job, and not between operations of different jobs.) For example, if we impose a strict total order on the order of execution of the operations of a job, the problem is a *job shop* scheduling problem. If the total order is the same total order for every job, and each job has at most one operation on each machine, we have a *flow shop* scheduling problem. If there is no order at all imposed on the execution of any job's operations, we have an *open shop* problem. It is traditional in the scheduling literature to focus, for the open shop problem, on the case when each job is processed on each machine at most once (since operations on the same machine can be coalesced). We will refer to the general shop scheduling problem that does not fall into one of the three above categories as the *dag shop* problem.

In this thesis we will concentrate primarily on the job shop scheduling problem, for two reasons. First of all, most of our results for other shop problems can be obtained as easy corollaries of our results for the job shop problem. Secondly, the job shop problem is probably the most famous and most difficult of all the versions of the problem. It is strongly $\mathcal{NP}$-hard; furthermore, except for the cases when there are two jobs or when there are two machines *and* each job has at most two operations, essentially all special cases of this problem are $\mathcal{NP}$-hard, and typically strongly $\mathcal{NP}$-hard [44, 81]. For example, it is $\mathcal{NP}$-hard even if there are 3 machines, 3 jobs and each operation is of unit length; note that in this case we can think of the input length as the maximum number of operations in a job, μ.

In addition to this theoretical evidence of the difficulty of the job shop problem, it is also one of the most notoriously difficult $\mathcal{NP}$-hard optimization problems in terms of practical computation, with even very small instances being difficult to solve exactly. A striking example of this is that a single instance of the problem involving only 10 jobs, 10 machines and 100 operations, which first appeared in a book by Muth and Thompson in 1963, remained unsolved for 23 years despite repeated attempts to find an optimal solution [81]. Today, due to better algorithms and faster machines, instances with 10 jobs and 10 machines seem to be tractable - Applegate and Cook solved ten different 10×10 problems, including the notorious instance mentioned above, in times ranging from 90 seconds to 42 minutes. (It is interesting to note that the instance of

Muth and Thompson was one of the easier instances to solve using their technique). However, slightly larger instances are still currently intractable; they report instances of size 10×15, 15×20, 15×15 and 10×20 that they were unable to solve [3].

Formal Definition and Previous Results

We formally define the job shop problem as follows. We are given a set $\mathcal{M} = \{m_1, m_2, \ldots, m_m\}$ of machines, a set $\mathcal{J} = \{J_1, \ldots, J_n\}$ of jobs, and a set $\mathcal{O} = \{O_{ij} | i = 1, \ldots, \mu_j, j = 1, \ldots, n\}$ of operations, where κ_{ij} indexes the machine on which operation O_{ij} runs. Thus m is the number of machines, n is the number of jobs, μ_j is the number of operations of job J_j, and $\mu = \max_j \mu_j$. O_{ij} is the ith operation of J_j; it requires processing time on a given machine $m_k \in \mathcal{M}$, where $k = \kappa_{ij}$, for an uninterrupted period of a given length p_{ij}. (In other words, this is a *non-preemptive* model; a model in which operations may be interrupted and resumed at a later time is called a *preemptive* model.) Each machine can process at most one operation at a time, and each job may be processed by at most one machine at a time. If the completion time of operation O_{ij} is denoted by C_{ij}, then the objective is to produce a schedule that minimizes the maximum-completion time, $C_{\max} = \max_{i,j} C_{ij}$; the optimal value is denoted by $C^*_{\max}$.

It is possible to extend this model by associating with each job J_j a *release date* r_j, on which J_j becomes available for processing. This extension does not affect most of our performance bounds, and therefore unless explicitly stated otherwise we assume that all jobs are available for processing at time 0.

The formal definition of the flow, open or dag shop problems would be almost the same, except for the following small differences:

- *flow shop:* $\kappa_{ij} = \kappa_{ij'}$, for all i, j, j', and $\kappa_{ij} \neq \kappa_{i'j}$ for all i, i', j.
- *open shop:* The O_{ij} can be processed in *any* order.
- *dag shop:* For each job j we define a partial order on the O_{ij} and require that they be processed in any total order consistent with that partial order.

There are two very easy lower bounds on the length of an optimum schedule. Since each job must be processed, $C^*_{\max}$ must be at least the maximum total length of any job, $\max_J \sum_i p_{ij}$, which we shall call the *maximum job length* of the instance, $P_{\max}$. Furthermore, each machine must process all of its operations, and so $C^*_{\max}$ must be at least $\max_{m_k} \sum_{\kappa_{ij}=k} p_{ij}$, which we will call the *maximum machine load* of the instance, $\Pi_{\max}$. Note that these are lower bounds regardless of whether we have a job, flow, open or dag shop problem.

There has been a tremendous amount of literature on shop scheduling problems over the last thirty years [81]. We mentioned earlier that all but the most restrictive versions of the job shop problem are $\mathcal{NP}$-hard; this is also true of the other versions of the problem. When there are at least 3 machines both the open and flow shop problems are $\mathcal{NP}$-hard [81]. When there are just two machines both these problems are known to be in $\mathcal{P}$ [66, 50]. In contrast, the two machine job shop problem is only known to be polynomial-time solvable if each job has at most two operations, or if each operation is of unit size [81].

Despite all the attention, however, surprisingly little has been known about approximation algorithms for shop scheduling problems. In fact, all that was known was the following observation by Gonzales and Sahni:

Theorem 1.1.1 [51] An algorithm $\mathcal{A}$ for the job shop problem that produces a schedule in which at least one machine is running at any point in time is an m-approximation algorithm.

Proof: The length of the schedule produced by such an algorithm $C_{\max}(\mathcal{A})$ is bounded above by $\sum_{i,j} p_{ij}$, since some operation is always being executed. On the other hand, the *average machine load*, $\sum_{i,j} p_{ij}/m$, is a lower bound on the *maximum machine load*, which is a lower bound. The theorem follows directly. ■

Little was also known in the way of negative results – results that indicate it is difficult to approximate these problems. Recently, however, David Williamson has shown that unless $\mathcal{P} = \mathcal{NP}$ none of these problems can be approximated arbitrarily closely.

Theorem 1.1.2 [124] Unless $\mathcal{P} = \mathcal{NP}$,

- There is no polynomial time algorithm that approximates the job shop problem or the flow shop problem within a factor of $\frac{13}{12}$.
- There is no polynomial time algorithm that approximates the open shop problem within a factor of $\frac{7}{6}$.

Despite the lack of knowledge about approximation algorithms with good worst-case relative error guarantees, there are two relevant results that are important to our work. The most interesting approximation algorithms to date for job shop scheduling have primarily appeared in the Soviet literature and are based on a beautiful connection to geometric arguments. This approach was independently discovered by Belov and Stolin [6] and Sevast'yanov [108] as well as by Fiala [39]. This approach typically produces schedules for which the length can be bounded by $\Pi_{\max} + q(m, \mu)p_{\max}$, where $q(\cdot,\cdot)$ is a polynomial, and $p_{\max} = \max_{ij} p_{ij}$ is the maximum operation length. For the job shop problem, Sevast'yanov [109, 110] gave a polynomial-time algorithm that delivers a schedule of length at most $\Pi_{\max} + O(m\mu^3)p_{\max}$. The bounds obtained in this way do not give good worst-case relative error bounds. Even for the special case of the *flow shop problem*, the best known algorithms delivered solutions of length $\Omega(mC^*_{\max})$.

Since these results are not well known in the West, and are important tools for us, we provide here a bit of information about the proof of the flow shop result, which is simpler than the more general job shop result. This simpler presentation of the proof is due to David Shmoys [114].

Theorem 1.1.3 There exists a polynomial time algorithm $\mathcal{A}$ for the flow shop problem which yields a schedule of length bounded above by $C^*_{\max} + m(m-1)p_{\max}$.

Proof:

The proof relies heavily on the following lemma.

Lemma 1.1.4 Let $\{v_1, v_2, \ldots, v_n\}$ be a set of d-dimensional vectors such that $\sum_{j=1}^{n} v_j = 0$. There exists a polynomial-time algorithm that computes a permutation π such that for any $k = 1, \ldots, n$, $||\sum_{j=1}^{k} v_{\pi(j)}|| \leq d \max_j ||v_j||$, where we use $||x||$ to denote the L_1-norm of x.

Without loss of generality, we can assume that the load on each machine is equal to the maximum machine load, namely $\Pi_{\max}$. In this case the completion time of the schedule is $\Pi_{\max} + I$, where I is the amount of idle time on the last machine before it starts processing the last operation of the last job to complete on it. If we choose a permutation π of the n jobs and schedule their operations in that order on every machine, it is not hard to see that the condition $\sum_{l=1}^{j}(p_{i-1,\pi(l)} - p_{i,\pi(l)}) \leq (m-1)p_{\max}$ would yield an upper bound of $m(m-1)p_{\max}$ on I.

Now if we construct a set of n m-dimensional vectors v_j, where $v_j = (p_{1j} - p_{2j}, p_{2j} - p_{3j}, \ldots, p_{m-1,j} - p_{mj})$, the algorithm mentioned in the previous lemma will produce the necessary permutation. ■

Another important result on shop scheduling comes, somewhat surprisingly, from the literature on packet routing. Leighton, Maggs and Rao [82] have proposed the following model for the routing of packets in a network: find paths for the packets and then schedule the transmission of the packets along these paths so that no two packets traverse the same edge simultaneously. The primary objective is to minimize the time by which all packets have been delivered to their destination.

It is easy to see that the scheduling problem considered by Leighton, Maggs and Rao is simply the job shop scheduling problem with each processing time $p_{ij} = 1$. They also added the restriction that each path does not traverse any edge more than once, or in scheduling terminology, each job has at most one operation on each machine. This restriction of the job shop problem remains (strongly) $\mathcal{NP}$-hard [81]. The main result of Leighton, Maggs and Rao was to show that for their special case of the job shop problem, there always exists a schedule of length $O(\Pi_{\max} + P_{\max})$. Unfortunately, this is not an algorithmic result, as it relies on a nonconstructive probabilistic argument based on the Lovász Local Lemma. They also obtained a randomized algorithm that delivers a schedule of length $O(\Pi_{\max} + P_{\max} \log n)$, with high probability.

Brief Statement of Our Results: We give a randomized $O(\frac{\log^2(m\mu)}{\log\log(m\mu)})$- approximation algorithm and a deterministic $O(\log^2(m\mu))$-approximation algorithm for the job shop scheduling problem, and a 2-approximation algorithm for open shop scheduling. All of these significantly improve on the previous best bounds of m. We also give two parallel algorithms: a $\mathcal{RNC}$ $O(\frac{\log^2(n\mu)}{\log log(n\mu)})$-approximation algorithm for job shop scheduling and an $\mathcal{NC}$ $O(\log n)$-approximation algorithm for open shop scheduling. Our results for job shop scheduling extend to a number of important generalizations of the basic job shop problem.

1.2 The Parallel Machine Environment

In contrast to the shop environment, where a job consists of multiple operations, each of which must be processed by a particular machine, in the *parallel machine environment* each job has only one operation, which may be processed by any machine. An instance consists of n jobs and m machines. Each machine can process at most one job at a time, and each job must be processed in an uninterrupted fashion on one of the machines. Typically one of three assumptions is made about the relative powers of the machines. In the most general setting, the machines are *unrelated*: job J_j takes $p_{ij} = p_j/s_{ij}$ time units when processed by machine m_i, where p_j is the *processing requirement* (or *size*) of job J_j and s_{ij} is the speed of machine m_i on job J_j. If the machines are *uniformly related*, then each machine m_i runs at a given speed s_i for all jobs J_j, and the processing time p_{ij} is given by p_j/s_i. Finally, for *identical* machines, we assume that $s_i = 1$ for each machine m_i. If C_j denotes the time at which job J_j completes processing in a schedule, then the *makespan* or *length* of the schedule is $C_{\max} = \max_j C_j$. For a given instance $\mathcal{I}$, our objective is, once again, to find a schedule of minimum length $C^*_{\max}(\mathcal{I})$.

In an off-line setting, these three types of parallel machine models have been studied extensively. The associated scheduling problems are all strongly $\mathcal{NP}$-hard [44], and polynomial approximation schemes are known when the machines are either identical or uniformly related [60, 61]. For unrelated machines, obtaining a solution better than $(3/2)C^*_{\max}$ is $\mathcal{NP}$-hard, whereas a schedule of length at most $2C^*_{\max}$ can be found in polynomial time [83]. We will be interested in *on-line* algorithms to solve these problems, algorithms that produce a good schedule without knowing the size of a job until it is finished being processed and not knowing what jobs will arrive in the future. We put off a discussion of the motivation for and precise definitions of on-line algorithms until Chapter 3.

We will also consider the *preemptive* versions of these models, in which a job may be interrupted on one machine and continued later (possibly on another machine) without penalty. In each of these three models, there is a polynomial-time off-line algorithm to find an optimal preemptive solution [89, 62, 79].

Brief Summary of Our Results: We introduce two rather general techniques for converting scheduling algorithms that need more complete knowledge of the input data into ones that need less advance knowledge. Using these techniques we give deterministic on-line scheduling algorithms for all the parallel machine models we have described. For all except the unrelated machines models we can prove that these algorithms are asymptotically optimal. We also give an on-line algorithm for the non-preemptive scheduling of uniformly related machines that takes advantage of the structure of the distribution of speeds amongst the machines, and we prove that this algorithm is asymptotically optimal as well. We also prove a surprisingly strong lower bound on the performance of any randomized on-line algorithm for the non-preemptive scheduling of identical machines.

1.3 Scheduling Problems and Parallel Computation

Efficiently scheduling a set of tasks on a set of machines is a basic and important problem in scheduling theory and in combinatorial optimization. It also, in some ways, captures the essence of the challenge of parallel computation. The models we have discussed in this section are the traditional models of combinatorial scheduling theory, but are simplifications of useful models for real parallel machines. We therefore present here a small sample of scheduling models that are related to those we consider but capture in more detail the problems involved in designing parallel algorithms and systems. We have discussed earlier the relationship between the shop scheduling model and packet-routing problems, so in this section we focus on the parallel machine model.

In a theoretical result, Papdimitriou and Yannakakis proposed a scheduling model as a tool for architecture-independent analysis of parallel algorithms [93]. They assert that the performance analysis of a parallel algorithm for a problem consists of at least four stages:(1) Choosing the algorithm, which is described by a directed acyclic graph (dag) that captures the elementary computations and their interdependence. The constraints imposed by the dag are often called *precedence constraints.* (2) Choosing a particular multiprocessor architecture. (3) Finding a *schedule* whereby the algorithm is executed on that architecture, so that the necessary data are available to the appropriate processor at the time of each computation. (4) At this point it is appropriate to discuss the performance of the algorithm, which is characterized by the makespan of the schedule.

They introduce the parameter τ, the communication delay between the time some information is produced at a processor and the time that it can be used at another processor. Assuming that there are enough processors to handle the width of the dag, they give an approximation algorithm that for any τ and any dag comes within a factor of 2 of the optimal makespan. Despite the assumption that disregards the number of processors the technique does not necessarily produce unrealistic processor-wasteful algorithms. In fact they are able to derive asymptotic upper and lower bounds on the parallel complexity of several algorithms that are processor-optimal for fixed time, and yield quite intuitive algorithms.

Berger and Cowen considered a more general model of program dependence than a simple dag. A dag captures only one sort of dependency: that computation i must be performed before computation j. Draper [30] gives examples where the ability to require *concurrent* scheduling of two computations or to require only that one computation is performed before or simultaneously with another operation would increase the ability to exploit the parallelism of a parallel system. They therefore consider the problem of scheduling unit-size jobs on m identical machines when the execution of the jobs can be constrained by all of the types of constraints they describe. Their results apply to a problem that arises in practice on the *Horizon* architecture [8].

Feitelson and Rudolph [36, 37] argue that a central theme of parallel processing is the existence of many interacting threads of control that cooperate to perform a single computation. The threads can be created throughout the computation and execute for different amounts of times. Since threads may need to interact there may be a need for synchronization; furthermore if the number of threads exceeds the number of processors a naive approach may leave threads sitting idle while they wait for other threads to be processed. A solution they suggest and are implementing is *gang scheduling*, where one guarantees that a set of interacting threads executes simultaneously. The model of gang scheduling is similar to our identical machine model, but jobs now also have a *width* which is the size of the block of processors on which the job must be executed. They argue that the scheduling must be done in an on-line fashion, and study a number of different rules under various distributions.

1.4 Notation Summary

Due to the large number of different and important versions of scheduling problems, there is a fair bit of notation associated with the field. We close this chapter with a list of the scheduling notation used in this thesis, organized in such a fashion as to summarize the similarities and differences between the different models.

List of Notation

Common to both environments:

- m: the number of machines.
- n: the number of jobs.
- $J_j, 1 \leq j \leq n$: the jth job.
- $m_i, 1 \leq i \leq m$: the ith machine.
- $\mathcal{I}$: a problem instance.
- $\mathcal{A}$: an algorithm.
- $C^*_{\max}(\mathcal{I})$ the length (makespan) of the shortest feasible schedule for instance $\mathcal{I}$.
- $C_{\max}(\mathcal{A})$: length (makespan) of the schedule produced by algorithm $\mathcal{A}$.

Shop Scheduling:

- O_{ij}: ith operation of job j.
- μ: the maximum number of operations in any job.
- κ_{ij}: the machine on which operation O_{ij} must be processed.
- $\Pi_{\max}$: the maximum machine load.
- p_{ij}: the size of operation O_{ij}.
- $p_{\max}$: the maximum operation size.

- $P_{\max}$: the maximum total job size.

Parallel Machine Scheduling:

- p_j: size of job j.
- s_i: speed of machine i (Uniformly related machines).
- s_{ij}: speed of machine i on job j (Unrelated machines).
- $p_{\max}$: the maximum job size.

As a final point of notation, except where otherwise specified $\log x$ will always refer to the logarithm base 2 of x.

Chapter 2

Approximation Algorithms for Shop Scheduling[1]

2.1 Introduction

As we have discussed in Chapter 1, despite over thirty years of work on shop scheduling, very little has been known about approximation algorithms for these problems. In this chapter we present the first nontrivial approximation algorithms for these scheduling problems. Our most important contribution is the first randomized polynomial-time polylogarithmic approximation algorithm for job shop scheduling. This algorithm builds on and generalizes the framework established by Leighton, Maggs and Rao for the special case of unit operation sizes and at most one operation per job per machine [82].

Theorem 2.1.1 There exists a polynomial-time randomized algorithm for job shop scheduling, that, with high probability, yields a schedule that is of length $O(\frac{\log^2(m\cdot\mu)}{\log\log(m\cdot\mu)}C^*_{\max})$.

We also give a deterministic version of this algorithm with almost the same performance guarantee.

Theorem 2.1.2 There exists a deterministic polynomial-time algorithm for job shop scheduling which finds a schedule of length $O(\log^2(m\cdot\mu)C^*_{\max})$.

[1]This chapter describes joint work with David Shmoys and Cliff Stein [112].

As a corollary, we also obtain a deterministic version of the randomized algorithm of Leighton, Maggs and Rao. Our deterministic algorithm relies on results of Raghavan and Thompson [102] and Raghavan [99] to approximate certain integer packing problems. Note that if each job must be processed on each machine at most once, the factor of μ can be deleted for this, and all other performance guarantees in this chapter.

Our techniques are not only useful for the job shop problem, but can easily be extended to the general problem of *dag shop scheduling.* Another important generalization is the situation where, rather than having m different machines, there are m' types of machines, and for each type, there are a specified number of identical machines; each operation, rather than being assigned to one machine, may be processed on any machine of the appropriate type. These problems have significant practical importance, since in real-world shops we would expect that a job need not follow a total order and that the shop would have more than one copy of many of their machines. We will give approximation algorithms with the same performance guarantees for this generalization as well.

When m and μ are constants we can achieve much better approximation guarantees – we give a $(2+\epsilon)$-approximation algorithm for this special case. Finally, we give *parallel* approximation algorithms for all the scheduling models mentioned above, and some improved results for the open shop problem.

While all of the algorithms that we give are polynomial-time, they are all rather inefficient. Most rely on the algorithms of Sevast'yanov; for example, his algorithm for job shop scheduling takes $O((\mu mn)^2)$ time. Furthermore, the deterministic versions rely on linear programming. As a result, we will not refer explicitly to running times throughout the remainder of this chapter.

The rest of this chapter is organized as follows. In Section 2.2 we extend the basic technique of Leighton, Maggs and Rao to the general job shop problem. In Section 2.3 we show how to scale and reduce the input data so that the techniques of Section 2.2 yield good performance bounds. In Section 2.4 we show how our techniques apply to more general problems, and in 2.5 we show how to make them deterministic. We conclude with a discussion of the open shop problem in Section 2.6 and some open problems in Section 2.7.

2.2 The Basic Algorithm

In this section we extend the technique due to Leighton, Maggs and Rao [82] of assigning random delays to jobs to the general case of non-preemptive job shop scheduling. A valid schedule assigns at most one job to a particular machine at any time, and schedules each job on at most one machine at any time. Their approach, for the special case of unit-size operations and at most one operation of each job on each machine, was to first create a schedule that obeyed only the second constraint, and then build from this a schedule that satisfies both constraints and is not much longer. The outline of the strategy follows:

1. Define the *oblivious* schedule, where each job starts running at time 0 and runs continuously until all of its operations have been completed. This schedule is of length $P_{\max}$, but there may be times when more than one job is assigned to a particular machine.

2. Perturb this schedule by delaying the start of the first operation of each job by a random integral amount chosen uniformly in $[0, \Pi_{\max}/\log n]$. The resulting schedule, with high probability, has no more than $O(\log n)$ operations assigned to any machine at any time.

3. Reschedule each unit of time t into $O(\log n)$ units of time during which each of the $O(\log n)$ operations scheduled for time t is processed. The resulting (valid) schedule is of length $O(P_{\max} \log n + \Pi_{\max})$.

Our strategy builds upon this framework of Leighton, Maggs and Rao. Whereas Step 2 differs in only a few technical details, the essential difficulty in obtaining the generalization is in Step 3.

2. Perturb this schedule by delaying the start of the first operation of each job by a random integral amount chosen uniformly in $[0, \Pi_{\max}]$. The resulting schedule, with high probability, has no more than $O(\frac{\log(n \cdot \mu)}{\log\log(n \cdot \mu)})$ jobs assigned to any machine at any time.

3. "Spread" this schedule so that at each point in time all operations currently being processed have the same size, and then "flatten" this into a schedule that has at most one job per machine at any time.

For the analysis of Step 2, we assume that $p_{\max}$ is bounded above by a polynomial in n and μ; in the next section we will show how to remove this assumption. As is usually the case, we assume that $n \geq m$; analogous bounds can be obtained when this is not true.

Lemma 2.2.1 Given a job shop instance in which $p_{\max}$ is bounded above by a polynomial in n and μ, the strategy of delaying each job an initial integral amount chosen randomly and uniformly from $[0, \Pi_{\max}]$ and then processing its operations in sequence will yield an (invalid) schedule that is of length at most $\Pi_{\max} + P_{\max}$ and, with high probability, has no more than $O(\frac{\log(n\cdot\mu)}{\log\log(n\cdot\mu)})$ jobs scheduled on any machine during any unit of time.

Proof: Fix a time t and a machine m_i; consider $p =$ Prob[at least τ units of processing are scheduled on machine i at time t]. There are at most $\binom{\Pi_{\max}}{\tau}$ ways to choose τ units of processing from all those required on m_i. If we focus on a particular one of these τ units and a specific time t, then the probability that it is scheduled at time t is at most $1/\Pi_{\max}$, since we selected a delay uniformly at random from among $\Pi_{\max}$ possibilities. If all τ units are from different jobs then the probability that they are all scheduled at time t is at most $(\frac{1}{\Pi_{\max}})^{\tau}$ since the delays are chosen independently. Other ise, the probability that all τ are scheduled then is 0, since it is impossible. Therefore

$$\begin{aligned} p &\leq \binom{\Pi_{\max}}{\tau}\left(\frac{1}{\Pi_{\max}}\right)^{\tau} \\ &\lesssim \left(\frac{e\Pi_{\max}}{\tau}\right)^{\tau}\left(\frac{1}{\Pi_{\max}}\right)^{\tau} \leq \left(\frac{e}{\tau}\right)^{\tau}. \end{aligned}$$

If $\tau = k\frac{\log(n\cdot\mu)}{\log\log(n\cdot\mu)}$ then $p < (n\mu)^{-(k-1)}$. To bound the probability that *any* machine at *any* time has more than $k\frac{\log(n\cdot\mu)}{\log\log(n\cdot\mu)}$ jobs using it, multiply p by $P_{\max} + \Pi_{\max}$ for the number of time units in the schedule, and by m for the number of machines. Since we have assumed that $p_{\max}$ is bounded by a polynomial in n and μ, $P_{\max} + \Pi_{\max}$ is as well; choosing k large enough yields that, with high probability, no more than $k\frac{\log(n\cdot\mu)}{\log\log(n\cdot\mu)}$ jobs are scheduled for any machine during any unit of time. ■

In the special case of unit-length operations treated by Leighton, Maggs and Rao, a schedule S of length L that has at most c jobs scheduled on any machine at any unit of time can trivially be "flattened" into a valid schedule of length cL by replacing one unit of S's time with c units

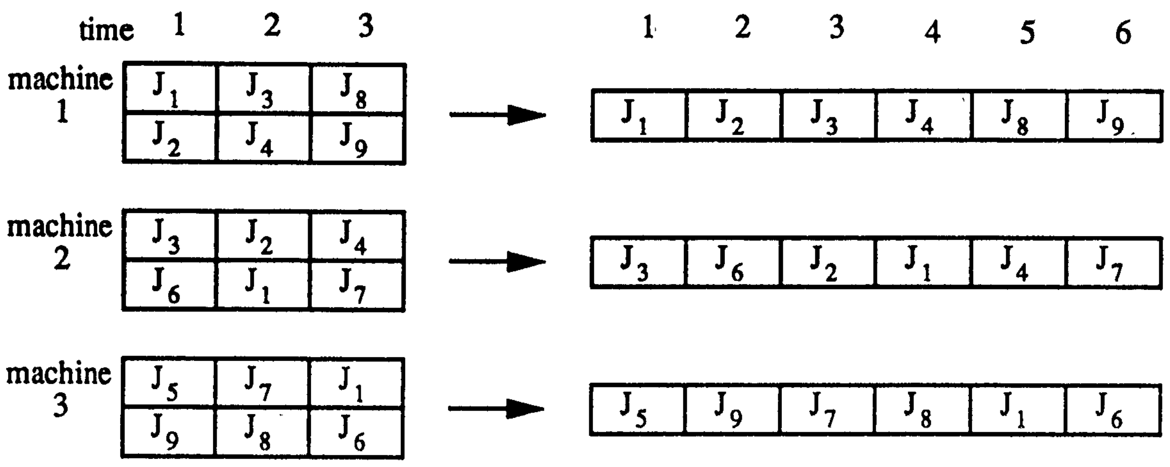

Figure 2.1: Flattening a schedule in the case with unit length operations.

of time in which we run each of the jobs that was scheduled for that time unit. (See Figure 2.1.)

For *preemptive* job shop scheduling, where the processing of an operation may be interrupted, each unit of an operation can be treated as a unit-length operation and a schedule that has multiple operations scheduled simultaneously on a machine can easily be flattened into a valid schedule. This is not possible for *non-preemptive* job shop scheduling, and in fact it seems to be more difficult to flatten the schedule in this case. We give an algorithm that takes a schedule of length L with at most c operations scheduled on one machine at any time and produces a schedule of length $O(cL \log p_{\max})$.

Lemma 2.2.2 Given a schedule $\mathcal{S}_0$ of length L that has at most c jobs scheduled on one machine during any unit of time, there exists a polynomial-time algorithm that produces a valid schedule of length $O(cL \log p_{\max})$.

Proof: To begin, we round up each processing time p_{ij} to the next power of 2 and denote the rounded times by p'_{ij}; that is, $p'_{ij} = 2^{\lceil \log_2 p_{ij} \rceil}$. Let $p'_{\max} = \max_{ij} p'_{ij}$. From $\mathcal{S}_0$, it is easy to obtain a schedule $\mathcal{S}$ that uses the modified p'_{ij} and is at most twice as long as $\mathcal{S}_0$; furthermore, an optimal schedule for the new problem is no more than twice as long as an optimal schedule for the original problem.

A *block* is an interval of a schedule with the property that each operation that begins during this interval is of length no more than that of the entire interval. (Note that this does not mean that the operation finishes within the interval.) We can divide $\mathcal{S}$ into $\lceil \frac{L}{p'_{\max}} \rceil$ consecutive blocks of size $p'_{\max}$. We will give a recursive algorithm that reschedules – "spreads" – each block of size p (where p is a power of 2) into a sequence of schedule *fragments* of total length $p \log p$; the operations scheduled in a fragment of length T are all of length T, and start at the beginning of the fragment. This algorithm takes advantage of the fact that if an operation of length p is scheduled to begin in a block of size p, then that job is not scheduled on any other machine until after this block. Therefore, that operation can be scheduled to start after all of the smaller operations in the block finish.

To reschedule a block B of size $p'_{\max}$, we first construct the final fragment (which is of length $p'_{\max}$), and then construct the preceding fragments by recursive calls of the algorithm. For each operation of length $p'_{\max}$ that begins in B, reschedule that operation to start at the beginning of the final fragment, and delete it from B. Now each operation that still starts in B is of length at most $p'_{\max}/2$, so B can be subdivided into two blocks, B_1 and B_2, each of size $p'_{\max}/2$, and we can recurse on each. See Figure 2.2.

The recurrence equation that describes the total length of the fragments produced from a block of size T is $f(T) = 2f(\frac{T}{2}) + T; f(1) = 1$. Thus $f(T) = \Theta(T \log T)$, and each block B in $\mathcal{S}$ of size $p'_{\max}$ is spread into a schedule of length $p'_{\max} \log p'_{\max}$. By spreading the schedule $\mathcal{S}$, we produce a new schedule $\mathcal{S}'$ that satisfies the following conditions:

1. At any time in $\mathcal{S}'$, all operations scheduled are of the same length; furthermore, any two operations either start at the same time or do not overlap.

2. If $\mathcal{S}$ has at most c jobs scheduled on one machine at any time, then this must hold for $\mathcal{S}'$ as well.

3. $\mathcal{S}'$ schedules a job on at most one machine at any time.

4. $\mathcal{S}'$ does not schedule the ith operation of job J_j until the first $i-1$ are completed.

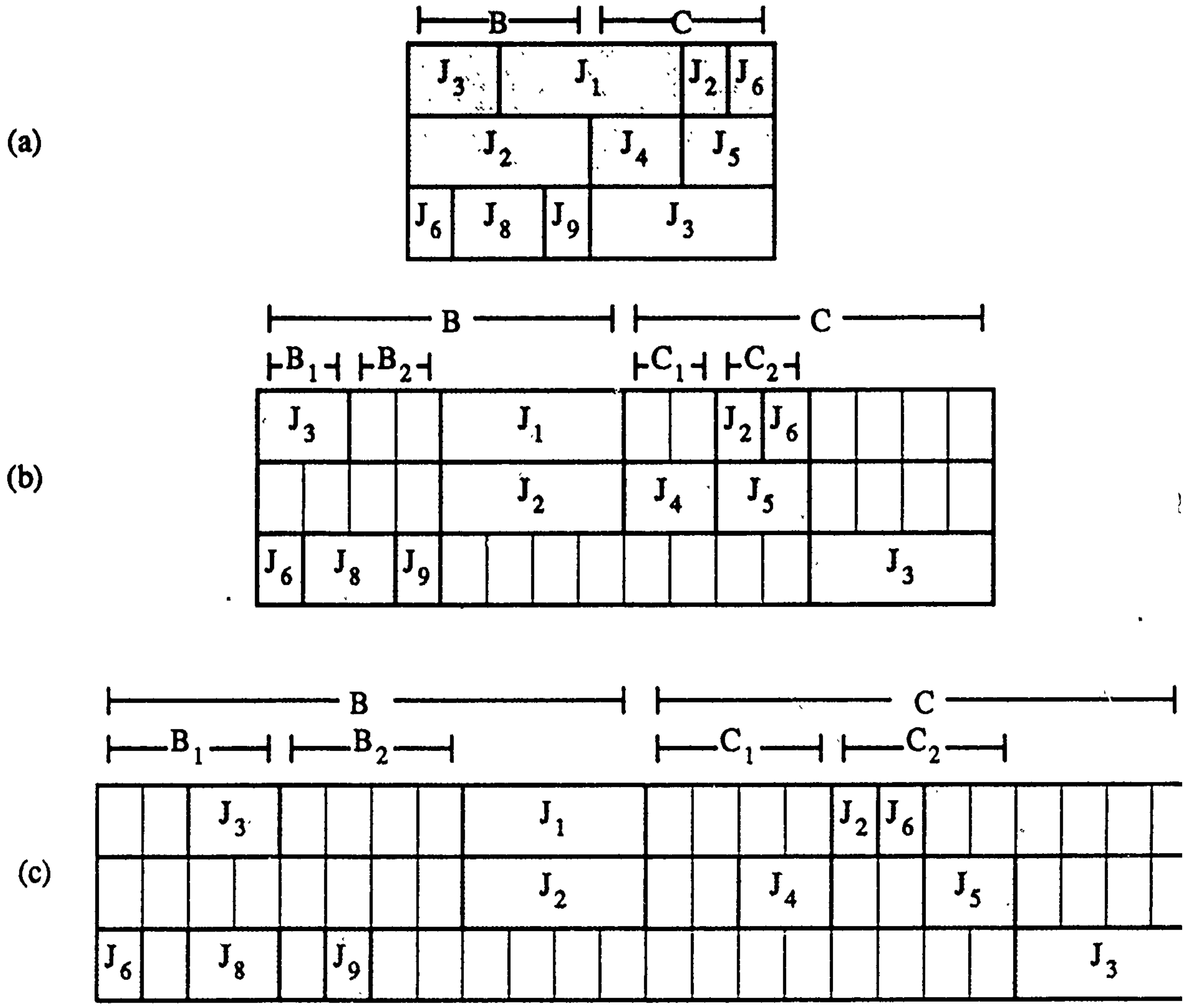

Figure 2.2: (a) The initial greedy schedule of length 8. $p'_{\max} = 4$. (b) The first level of spreading. All jobs of length 4 have been put in the final fragments. We must now recurse on B_1 and B_2 with $p'_{\max} = 2$. (c) The final schedule of length $8 \log_2 8 = 24$.

Condition 1 is satisfied by each pair of operations on the same machine by the definition of spreading, and by each pair of operations on different machines because the division of time into fragments is the same on all machines. To prove condition 2, note that operations of length T that are scheduled at the same time on the same machine in the expanded schedule started in the same block of size T on that machine. Since they all must have been scheduled during the last unit of time of that block, there can be at most c of them.

To prove condition 3, note that if a job is scheduled by $\mathcal{S}'$ on two machines simultaneously that means that it must have been scheduled by $\mathcal{S}$ to start two operations of length T in the same block of length T on two different machines. This means it was scheduled by $\mathcal{S}$ on two machines during the last unit of time of that block, which violates the properties of $\mathcal{S}$.

Finally we verify condition 4 by first noting that if two operations of a job are in different blocks of size $p'_{\max}$ in $\mathcal{S}$ then they are certainly rescheduled in the correct order. Therefore it suffices to focus on the schedule produced from one block. Within a block, if an operation is rescheduled to the final fragment then it is the last operation for that job in that block. Therefore $\mathcal{S}'$ does not schedule the ith operation of job J_j until the first $i-1$ are completed.

The schedule $\mathcal{S}'$ can easily be flattened to a schedule that obeys the constraint of one job per machine at any time, since c operations of length T that start at the same time can just be executed one after the other in total time cT. Note that since what we are doing is effectively synchronizing the entire schedule block by block, it is important when flattening the schedule to make each machine wait enough time for all machines to process all operations of that fragment length, even if some machines have no operations of that length in that fragment.

The schedule $\mathcal{S}'$ was of length $L \log p'_{\max}$; therefore the flattened schedule is of length $Lc \log p'_{\max}$. ■

We note in passing that the inclusion of release dates into the problem will not affect the quality of our bounds at all. The release dates can either be directly included into probabilistic analysis of lemma 2.2.1, or we can view each release date as one additional initial operation on some (imaginary) machine.

2.3 Reducing the Problem

In the previous section we showed how to produce, with high probability, a schedule of length

$$O\left((\Pi_{\max} + P_{\max})\frac{\log(n \cdot \mu)}{\log\log(n \cdot \mu)} \log p_{\max}\right),$$

under the assumption that $p_{\max}$ was bounded above by a polynomial in n and μ. Since

$$\Pi_{\max} + P_{\max} = O(\max\{\Pi_{\max}, P_{\max}\})$$

this schedule is within a factor of $O(\frac{\log(n \cdot \mu)}{\log\log(n \cdot \mu)} \log p_{\max})$ of optimality. In this section, we will first remove the assumption that $p_{\max}$ is bounded above by a polynomial in n and μ by showing that we can reduce the general problem to that special case while only sacrificing a constant factor in the approximation. This yields an $O(\frac{\log^2(n \cdot \mu)}{\log\log(n \cdot \mu)})$–approximation algorithm. Then we will show that n need not be polynomially bounded in m and μ. Combining these two results, we conclude that we can reduce the job shop problem to the case where n is polynomially bounded in m and μ, while changing the performance guarantee by only a constant.

2.3.1 Reducing $p_{\max}$

First we will show that we can reduce the problem to one where $p_{\max}$ is bounded by a polynomial in n and μ. Let $\omega = |\mathcal{O}|$ be the total number of required operations. Note that $\omega \leq n\mu$. Round down each p_{ij} to the nearest multiple of $p_{\max}/\omega$, denoted by p'_{ij}. Now there are at most ω distinct values of p'_{ij} and they are all multiples of $p_{\max}/\omega$. Therefore we can treat the p'_{ij} as integers in $\{0, \ldots, \omega\}$; a schedule for this problem can be trivially rescaled to a schedule S' for the actual p'_{ij}. (Note that assigning $p'_{ij} = 0$ does not mean that this operation does not exist; instead, it should viewed as an operation that takes an arbitrarily small amount of time.) Let L denote the length of S'. We claim that S' for this reduced problem can be interpreted as a schedule for the original operations that will be of length at most $L + p_{\max}$. When we adjust the p'_{ij} up to the original p_{ij}, we add an amount that is at most $p_{\max}/\omega$ to each p'_{ij}. Since the length of a schedule is determined by a critical path through the operations and there are ω operations, we add a total amount of at most $p_{\max}$ to the length of the schedule; thus the new schedule is of length at most $L + p_{\max} \leq L + C^*_{\max}$. Therefore we have rounded a general instance $\mathcal{I}$ of the job shop problem to an instance $\mathcal{I}'$ that can be treated as having $p_{\max} = O(n\mu)$; further, a schedule for $\mathcal{I}'$ yields a schedule for $\mathcal{I}$ that is no more than $C^*_{\max}$ longer. Thus we have shown:

Lemma 2.3.1 There exists a polynomial-time algorithm which transforms any instance of the job shop scheduling problem into one with $p_{\max} = O(n\mu)$ with the property that a schedule for the modified instance of length $kC^*_{\max}$ can be converted in polynomial time to a schedule for the original instance of length $(k+1)C^*_{\max}$.

2.3.2 Reducing the Number of Jobs

To reduce an arbitrary instance of job shop scheduling to one with a number of jobs polynomial in m and μ we divide the jobs into big and small jobs. We say that job J_j is *big* if it has an operation of length more than $\Pi_{\max}/(2m\mu^3)$; otherwise we call the job *small*. For the instance consisting of just the short jobs, let $\Pi'_{\max}$ and $p'_{\max}$ denote the maximum machine load and operation length, respectively. Using the algorithm of [110] described in the introduction, we can, in time polynomial in the input size, produce a schedule of length $\Pi'_{\max} + 2m\mu^3 p'_{\max}$ for this instance. Since $p'_{\max}$ is at most $\Pi_{\max}/(2m\mu^3)$ and $\Pi'_{\max} \leq \Pi_{\max}$, we get a schedule that is of length no more than $2\Pi_{\max}$. Thus, an algorithm that produces a schedule for the long jobs that is within a factor of k of optimal will yield a $(k+2)$-approximation algorithm. Note that there can be at most $2m^2\mu^3$ long jobs, since otherwise there would be more than $m\Pi_{\max}$ units of processing to be divided amongst m machines, which contradicts the definition of $\Pi_{\max}$. Thus we have shown:

Lemma 2.3.2 There exists a polynomial-time algorithm which transforms any instance of the job shop scheduling problem into one with $O(m^2\mu^3)$ jobs with the property that a schedule for the modified instance of length $kC^*_{\max}$ can be converted in polynomial time to a schedule for the original instance of length $(k+2)C^*_{\max}$.

From the results of the previous two sections we can conclude that:

Theorem 2.3.3 There exists a polynomial-time randomized algorithm for job shop scheduling, that, with high probability, yields a schedule that is of length at most $O(\frac{\log^2(m\cdot\mu)}{\log\log(m\cdot\mu)}C^*_{\max})$.

Proof: In Section 2 we showed how to produce a schedule of length

$$O\left((\Pi_{\max} + P_{\max})\frac{\log(n\cdot\mu)}{\log\log(n\cdot\mu)}\log p_{\max}\right)$$

under the assumption that $p_{\max}$ was bounded above by a polynomial in n and μ. From Lemmas 2.3.1 and 2.3.2 we know that we can reduce the problem to one where n and $p_{\max}$ are polynomial in m and μ, while adding only a constant to the factor of approximation. Since now $\log p_{\max} = O(\log(m\mu))$ and $\log n = O(\log(m\mu))$ our algorithm produces a schedule of length $O(\frac{\log^2(m\cdot\mu)}{\log\log(m\cdot\mu)}C^*_{\max})$. ■

Note that when μ is bounded by a polynomial in m the bound only depends on m. In particular, this implies the following corollary:

Corollary 2.3.4 There exists a polynomial-time randomized algorithm for flow shop scheduling, that, with high probability, yields a schedule that is of length at most $O(\frac{\log^2 m}{\log\log m}C^*_{\max})$.

Except for the use of Sevast'yanov's algorithm, all of these techniques can be carried out in $\mathcal{RNC}^2$. We assign one processor to each operation. The rounding in the proof of Lemma 2.2.2 can be done in $\mathcal{NC}$. We set the random delays and inform each processor about the delay of its job. By summing the values of p_{ij} for all of its job's operations, each processor can calculate where its operation is scheduled with the delays and then where it is scheduled in the recursively spread out schedule. These sums can be calculated via parallel prefix operations. With simple $\mathcal{NC}$ techniques we can assign to each operation a rank among all those operations that are scheduled to start at the same time on its machine, and thus flatten the spread out schedule to a valid schedule.

Corollary 2.3.5 There exists a $\mathcal{RNC}$ algorithm for job shop scheduling, that, with high probability, yields a schedule that is of length at most $O(\frac{\log^2(n\mu)}{\log\log(n\mu)}C^*_{\max})$.

[2]Since most of our discussion of parallel algorithms for combinatorial problems occurs in the latter chapters of this thesis, we have put off a discussion of $\mathcal{NC}$, $\mathcal{RNC}$ and models of parallel algorithms until Chapter 4. We refer the reader who is not familiar with these terms to that discussion.

2.3.3 A Fixed Number of Machines

It is interesting to note that Sevast'yanov's algorithm for the job shop problem can be viewed as a $(1 + m\mu^3)$-approximation algorithm, so that when m and μ are constant, this is a $O(1)$-approximation algorithm; that is, it delivers a solution within a constant factor of the optimum. The technique of partitioning the set of jobs by size can be applied to give a much better performance guarantee in this case. Now call a job J_j *big* if there is an operation O_{ij} with $p_{ij} > \epsilon\Pi_{\max}/(m\mu^3)$, where ϵ is an arbitrary positive constant. Note that there are at most $m^2\mu^3/\epsilon$ big jobs, and since m, μ and ϵ are fixed, this is a constant.

Now use Sevast'yanov's algorithm to schedule all of the small jobs. The resulting schedule will be of length at most $(1+\epsilon)C^*_{\max}$. There are only a constant (albeit a huge constant) number of ways to schedule the big jobs. Therefore the best one can be selected in polynomial time and executed after the schedule of the short jobs. The additional length of this part is no more than $C^*_{\max}$.

Thus we have shown:

Theorem 2.3.6 For the job shop scheduling problem where both m and μ are fixed, there is a polynomial-time algorithm that produces a schedule of length $\leq (2 + \epsilon)C^*_{\max}$.

2.4 Applications to More General Scheduling Problems

The fact that the quality of our approximations is based solely on the lower bounds $\Pi_{\max}$ and $P_{\max}$ makes it quite easy to extend our techniques to the more general problem of *dag shop scheduling*. We define $\Pi_{\max}$ and $P_{\max}$ exactly the same way, and $\max\{\Pi_{\max}, P_{\max}\}$ remains a lower bound for the length of any schedule. We can convert this dag shop scheduling problem to a job shop problem by selecting for each job an arbitrary total order that is consistent with its partial order. $\Pi_{\max}$ and $P_{\max}$ have the same values for both problems. Therefore, a scheduie of length $\rho \cdot (\Pi_{\max} + P_{\max})$ for this job shop instance is a schedule for the original dag shop scheduling instance of length $O(\rho C^*_{\max})$.

A further generalization to which our techniques apply is where, rather than m different machines, we have m' types of machines, and for each type we have a specified number of identical machines of that type. Instead of requiring an operation to run on a particular machine, an operation now has to run on only one of these identical copies. $P_{\max}$ remains a lower bound on the length of any schedule for this problem. $\Pi_{\max}$, which was a lower bound for the job shop problem must be replaced, since we do not have a specific assignment of operations to machines, and the sum of the processing times of all operations assigned to a type is *not* a lower bound. Let S_i, $i = 1, \dots m'$, denote the sets of identical machines, and let $\Pi(S_i)$ be the sum of the lengths of the operations which run on S_i. Our strategy is to convert this to a job shop problem by assigning operations to specific machines in such a way that the maximum machine load is within a constant factor of the fundamental lower bounds for this problem. To obtain a lower bound on the maximum machine load, note that the best we could do would be to evenly distribute the operations across machines in a set, thus

$$\Pi_{\text{avg}} = \max_{S_i} \frac{\Pi(S_i)}{|S_i|}$$

is certainly a lower bound on the maximum machine load. Furthermore, we can not split operations, so $p_{\max}$ is also a lower bound. We will now describe how to assign operations to machines so that the maximum machine load of the resulting job shop scheduling problem is at most $2\Pi_{\text{avg}} + p_{\max}$. A schedule for the resulting job shop problem of length $\rho \cdot (\Pi_{\max} + P_{\max})$ yields a solution for the more general problem of length $O(\rho \cdot (\Pi_{\text{avg}} + P_{\max}))$. Sevast'yanov [110] used a somewhat more complicated reduction to handle a slightly more general setting.

For each operation O_{ij} to be processed by a machine in S_k, if $p_{ij} \geq \Pi(S_k)/|S_k|$, assign O_{ij} to one machine in S_k. There are certainly enough machines in S_k to do this and this contributes at most $p_{\max}$ to the maximum machine load. Those operations not yet assigned are each of length at most $\Pi(S_k)/|S_k|$ and have total length $\leq \Pi(S_k)$. Therefore, these can be assigned easily to the remaining machines so that less than $2\Pi(S_k)/S_k$ processing units are assigned to each machine. Combining these two bounds, we get an upper bound on the maximum machine load of $2\Pi_{\text{avg}} + p_{\max}$ which is within a constant factor of the lower bound of $\max\{\Pi_{\text{avg}}, p_{\max}\}$.

Theorem 2.4.1 There exists a polynomial-time randomized algorithm for dag shop scheduling with identical copies of machines that, with high probability, yields a schedule that is of length at most $O(\frac{\log^2(m \cdot \mu)}{\log\log(m \cdot \mu)} C^*_{\max})$.

Corollary 2.4.2 There exists an $\mathcal{RNC}$ algorithm for dag shop scheduling with identical copies of machines that, with high probability, yields a schedule that is of length at most $O(\frac{\log^2(n\cdot\mu)}{\log\log(n\cdot\mu)}C^*_{\max})$.

2.5 A Deterministic Approximation Algorithm

In this section, we "derandomize" the results of the previous sections, i.e., we give a deterministic polynomial-time algorithm that finds a schedule of length $O(\log^2(m\cdot\mu)C^*_{\max})$. Of all the components of the algorithm of Theorem 2.3.3, the only step which is not already deterministic is the step that chooses a random initial delay for each job and then proves that, with high probability, no machine is assigned too many jobs at any one time. In particular, the reduction to the special case in which n and $p_{\max}$ are bounded by a polynomial in m and μ is entirely deterministic, and so we can focus on that case alone. We will give an algorithm which deterministically assigns delays to each job so as to produce a schedule in which each machine has $O(\log(m\mu))$ jobs running at any one time. We then apply Lemma 2.2.2 to produce a schedule of length $O(\log^2(m\cdot\mu)C^*_{\max})$. Note that the $O(\log(m\mu))$ jobs per machine is not as good as the probabilistic bound of $O(\frac{\log(m\cdot\mu)}{\log\log(m\cdot\mu)})$; we do not know how to achieve this deterministically. However, by a proof nearly identical to that of Lemma 2.2.1, we can show that in order to achieve this weaker bound on the number of jobs per machine, we now only need to choose delays in the range $[0, P_{\max}/\log(m\cdot\mu)]$. In fact, the reduced range of delays yields a schedule of length $O(P_{\max}\log^2(m\mu)+\Pi_{\max}\log(m\mu))$ which is within an $O(\log(m\mu))$ factor of optimal if $P_{\max}=O(\Pi_{\max}/\log(m\mu))$.

Our approach to solving this problem is to frame it as a *vector selection* problem and then apply techniques developed by Raghavan and Thompson [101, 102] and Raghavan [99] which find constant factor approximations to certain "packing" integer programs. The approach is to formulate the problem as a $\{0,1\}$–integer program, solve the linear programming relaxation, and then randomly round the solution to an integer solution.

For certain types of problems this yields provably good approximations with high probability [101, 102]. Furthermore, for many of the problems for which there are approximations with high probability, the algorithm can be derandomized. Raghavan [99] has shown how to do this by essentially setting the random bits one at a time.

We now state the problem formally:

Problem 2.5.1 Deterministically assign a delay to each job in the range $[0, P_{\max}/\log(m \cdot \mu)]$ so as to produce a schedule with no more than $O(\log(m\mu))$ jobs on any machine at any time.

Lemma 2.5.2 Problem 2.5.1 can be solved in deterministic polynomial time.

Proof: Since we introduce delays in the range $[0, P_{\max}/\log(m \cdot \mu)]$, the resulting schedule has length $\ell = P_{\max} + \Pi_{\max}/\log(m\mu)$. We can represent the processing of a job j with a given initial delay d by an $(\ell \cdot m)$-length $\{0,1\}$-vector where each position corresponds to a machine at a particular time. The position corresponding to machine m_i and time t is 1 if m_i is processing job J_j at time t, and 0 otherwise. For each job J_j and each possible delay d, there is a vector $V_{j,d}$ which corresponds to assigning delay d to J_j.

Let λ_j be the set of vectors $\{V_{j,1}, \ldots, V_{j,d_{\max}}\}$, where $d_{\max} = \Pi_{\max}/\log(m\mu)$, and let $V_{j,k}(i)$ be the i^{th} component of $V_{j,k}$. Given the set $\Lambda = \{\lambda_1, \ldots, \lambda_n\}$ of sets of vectors, our problem can be stated as the problem of choosing one vector from each λ_j (denoted V_j^*), such that

$$\left\| \sum_{j=1}^{n} V_j^* \right\|_\infty = O(\log(m\mu)),$$

i.e., at any time on any machine, the number of jobs using that machine is $O(\log(m\mu))$.

As in [99], we can reformulate this as a $\{0,1\}$-integer program. Let $x_{j,k}$ be the indicator variable used to indicate whether $V_{j,k}$ is selected from λ_j. Consider the integer program (IP) that assigns $\{0,1\}$ values to the variables $x_{j,k}$ to minimize W subject to the constraints:

$$\begin{aligned} \sum_{k=1}^{d_{\max}} x_{j,k} &= 1, \quad j = 1, \ldots, n, \\ \sum_{j=1}^{n} \sum_{k=1}^{d_{\max}} x_{j,k} V_{j,k}(i) &\le W, \quad i = 1, \ldots, \ell \cdot m. \end{aligned}$$

Let W_{OPT} be the optimum value of W, which is the maximum number of jobs that ever use a machine at any time. We already know, by Lemma 2.2.1, that $W_{\text{OPT}} = O(\log(m\mu))$, so if we could solve this integer program optimally we would be done. However, the problem is

$\mathcal{NP}$-hard. Instead, we rely on the following theorem which is immediate from the results in [99] and [102].

Theorem 2.5.3 [99, 102] A feasible solution to (IP) with $W = O(W_{\text{OPT}} + \log(m\mu))$ can be found in polynomial time.

■

We then apply Lemma 2.2.2 and obtain the following result:

Theorem 2.5.4 There exists a deterministic polynomial-time algorithm which finds a schedule of length $O(\log^2(m \cdot \mu) C^*_{\max})$.

2.6 The Open Shop Problem

Recall that in the open shop problem the operations of a job can be executed in any order. Fiala [40] has also shown that if $\Pi_{\max} \geq (16m \log m + 21m)p_{\max}$, then $C^*_{\max}$ is just $\Pi_{\max}$, and there is a polynomial-time algorithm to find an optimal schedule, but in general this problem is strongly $\mathcal{NP}$-Complete. We will show that, in contrast to the job and flow shop problems, a simple greedy strategy yields a fairly good approximation to the optimal open shop schedule. Consider the algorithm that, whenever a machine is idle, assigns to it any job that has not yet been processed on that machine and is not currently being processed on another machine. Anná Racsmány [5] has observed that the greedy algorithm delivers a schedule of length at most $\Pi_{\max} + (m-1)p_{\max}$. We can adapt her proof to show that, in fact, the greedy algorithm delivers a schedule that is no longer than a factor of two times optimal. We can also show that this is essentially a tight bound.

Theorem 2.6.1 The greedy algorithm for the open shop problem is a 2-approximation algorithm. This is true even when each job J_j has an associated *release date* r_j on which it becomes available for processing. Furthermore, this strategy can produce schedules that are as long as $(2 - \frac{1}{m})$ times optimal.

Proof: Consider the machine m_k that finishes last in the greedy schedule; this machine is active sometimes, idle sometimes, and finishes by completing some job J_j. Since the schedule is greedy, whenever m_k is idle, J_j is either being processed by some other machine or has not yet been released. Therefore, the idle time is at most $\sum_{m_i \neq m_k} p_{ij} + r_j < P_j + r_j$. Thus, machine m_k is processing for at most $\Pi_{\max}$ units of time and is idle for less than $P_j + r_j$ units of time; hence $C_{\max} < \Pi_{\max} + P_j + r_j$. However, $P_j + r_j$ is a lower bound on the length of the schedule, since no processing of job J_j could start until time r_j. Therefore the schedule length is within a factor of 2 of optimal.

To lower bound the worst-case performance of this algorithm, consider an open shop instance with $n = 2m - 1$ jobs. Job J_1 has one operation of size 1 on each machine, job J_2 has one operation of size $m - 1$ on machine 1 and job J_3 has one operation of size $m - 1$ on machine m. Finally, for each machine i, $2 \leq i \leq m - 1$, there are two jobs, each with one operation of machine i. One job has an operation of size $i - 1$ and the other job has an operation of size $m - i$. The optimal schedule is of length m but a greedy algorithm can produce a schedule of length $2m - 1$ (see Figure 2.6). ■

Using a slightly different (non-greedy) strategy, we can derive another algorithm which achieves a schedule of length $O(\log n C^*_{\max})$. This algorithm will also be easily parallelizable, thus putting the problem of finding an $O(\log n)$-approximation to the open shop scheduling problem in $\mathcal{NC}$.

We define the *jobs graph*, which is a bipartite graph that represents an instance of the open shop problem. One side of the bipartition contains m nodes, one for each machine, whereas the other side contains n nodes, one for each job. If job J_j has an operation on machine i then the jobs graph contains an edge between the respective nodes.

First consider the case when all the operations are of the same size, say ℓ. Let Δ be the maximum degree of any node in the remaining jobs graph. Then $\ell\Delta$ is a lower bound on the length of the optimal schedule for this problem. However, since this is a bipartite graph with maximum degree Δ, it can be edge-colored using exactly Δ colors. So we edge-color the graph, and then schedule the operations in each color class separately. This produces a schedule of length $\ell\Delta$, which is optimal. As long as there is at least one processor per operation, this can be done in NC using the edge-coloring algorithm of Lev, Pippinger, and Valiant [84].

We can extend this to solve the open shop problem by first using the techniques of Section 2.3.1 to reduce the problem to the case where all the operations have sizes polynomial in n, and

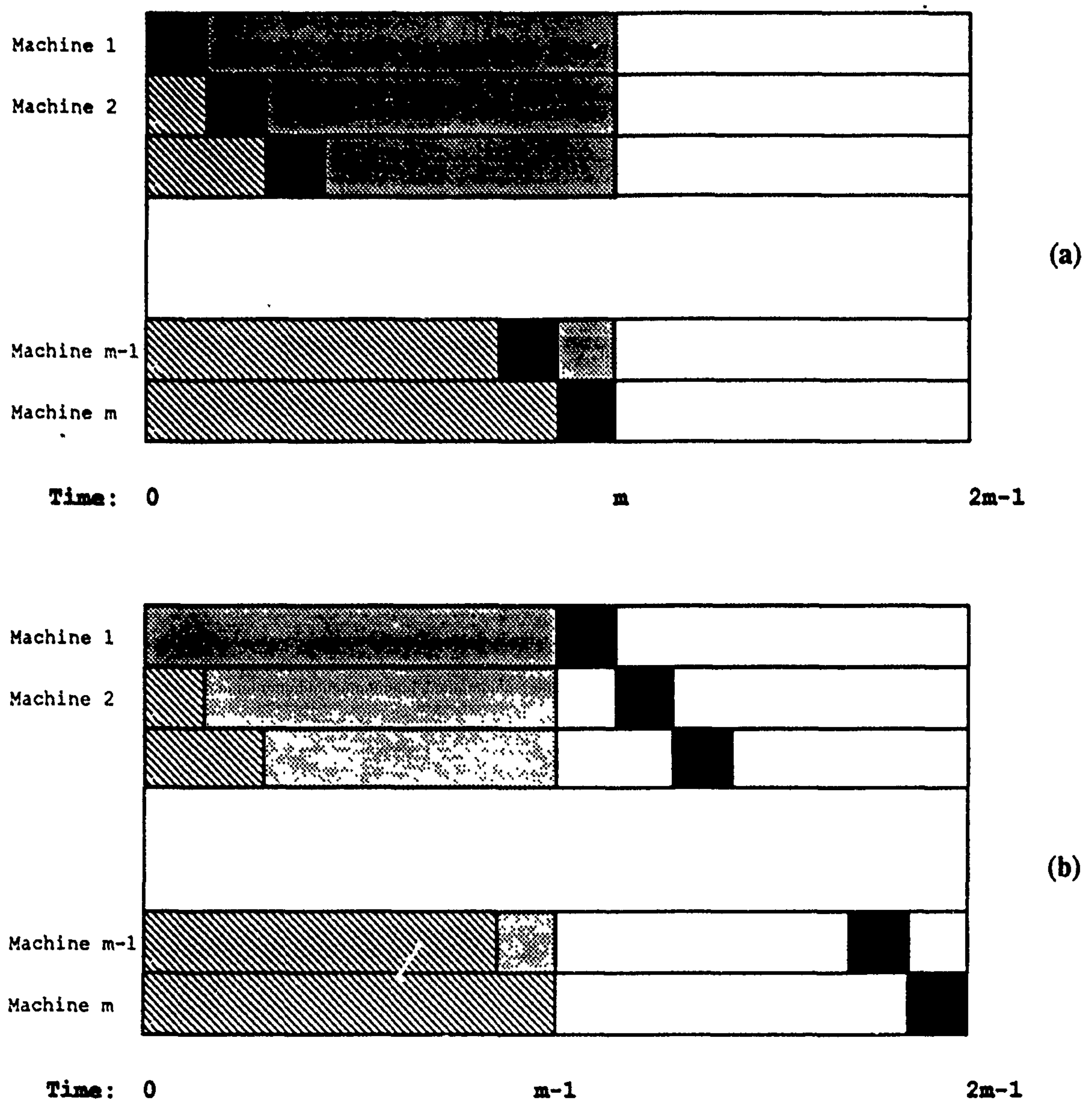

Figure 2.3: Instance for lower bound on performance of greedy open shop algorithm. Black squares represent the operations of job J_1. Schedule (a) is the optimum schedule, but schedule (b) is a greedy schedule in which J_1 is not started until after time $m-1$.

then by rounding the operation sizes so they are all powers of 2. Now there are only $O(\log n)$ different operation sizes. We schedule each one separately, using the edge-coloring strategy above. The schedule we get for any particular ℓ is optimal for that operations of that size; hence each of the $O(\log n)$ schedules we produce is of length $O(C^*_{\max})$. Concatenating these schedules together, and observing that all the rounding can easily be done in $\mathcal{NC}$, we obtain the following theorem:

Theorem 2.6.2 An open shop schedule of length $O(\log n C^*_{\max})$ can be found in $\mathcal{NC}$.

2.7 Conclusions and Open Problems

We have given the first polynomial-time polylog-approximation algorithms for minimizing the maximum completion time for the problems of job shop scheduling, flow shop scheduling, dag shop scheduling and a generalization of dag shop scheduling in which there are groups of identical machines. Clearly the most basic question to be pursued is the development of approximation algorithms with even better performance guarantees. It is our belief that the $O(\log p_{\max})$ factor that is introduced by the techniques of section 2.2 can be improved upon, perhaps even by a simple greedy method. However, such methods have proved frustratingly difficult to analyze. The other logarithmic factor in the performance bound seems much more difficult to improve upon.

An interesting consequence of our results is the following observation about the structure of shop scheduling problems. Assume we have a set of jobs which need to run on a set of machines. We know that any schedule for the associated open shop problem must be of length $\Omega(\Pi_{\max}+P_{\max})$. Furthermore, we know that no matter what type of partial ordering we impose on the operations of each job we can produce a schedule of length $O((\Pi_{\max}+P_{\max})\frac{\log^2 m}{\log\log m})$. Hence for any instance of the open shop problem, we can impose an arbitrary partial order on the operations of each job and increase the length of the optimal schedule by a factor of no more than $O(\frac{\log^2 m}{\log\log m})$.

An interesting combinatorial question is "Can this imposition of a partial order really make the optimal schedule that much longer than $O(\Pi_{\max}+P_{\max})$?" In other words, how good are $\Pi_{\max}$ and $P_{\max}$ as lower bounds? We have seen that in two interesting special cases, job shop scheduling with unit-length operations and open shop scheduling, there is a schedule of length $O(\Pi_{\max} + P_{\max})$. Does there always exist an $O(\Pi_{\max} + P_{\max})$ schedule for the general job, flow or dag shop scheduling problem?

Beyond this, there are a number of interesting questions raised by this work, including

- Do there exist parallel algorithms that achieve the approximations of our sequential algorithms? For the general job shop problem this seems hard, since we rely heavily on the algorithm of Sevast'yanov. For open shop scheduling, however, a simple sequential algorithm achieves a factor of 2, whereas the best $\mathcal{NC}$ algorithm that we have achieves only an $O(\log n)$-approximation. As a consequence of the results above, all one would need to do is to produce any greedy schedule.

- Are there simple variants of the greedy algorithm for open shop scheduling that achieve better performance guarantees? For instance, how good is the algorithm that always selects the job with the maximum total (remaining) processing time?

- Our algorithms, while polynomial-time algorithms, are inefficient. Are there significantly more efficient algorithms which have the same performance guarantees? Recent work by Plotkin, Shmoys and Tardos [97] gives a significantly faster algorithm, which does not use linear programming, to accomplish the derandomization described in Section 2.5. Therefore the major remaining problem is to develop a faster version of the algorithm of Sevast'yanov.

Chapter 3

Scheduling Parallel Machines On-line[1]

3.1 Introduction

The scheduling of a set of tasks on parallel machines[2] is a basic problem in combinatorial optimization, with an number of increasingly important applications. As we discussed in Chapter 1 there is a rich literature on parallel machine scheduling, but the overwhelming majority of these results assume that a complete specification of the instance is available before the algorithm begins to construct a schedule. This fails, however, to capture many of the scheduling problems that arise in practice. Consider, for example, the allocation of jobs to the processing units of a multiprocessor: the scheduler does not in advance have complete knowledge of a job's running time, or of what jobs will be created and require processing in the future. Or, consider the owner of a garage, who must schedule his group of car repairmen. The owner does not know how many cars will arrive to be repaired on a given day, and also does not know how long it will take to repair any particular car. In this chapter, we will study *on-line algorithms* – algorithms that work without any clairvoyant assumptions – for the most basic types of parallel machine scheduling models. Our algorithmic results are based on two rather general techniques that

[1]This chapter describes joint work with David Shmoys and David Williamson [113].

[2]See Chapter 1 for the definition of the parallel machine model and the associated notation.

allow us to convert algorithms that need more complete knowledge of the input data into ones that need less advance knowledge.

When on-line scheduling has been studied in the past, the models that have been considered were typically of the following form: the existence of a job is unknown until a certain *release date*, at which point the processing requirement for that job is completely specified. We will consider more realistic models, where the processing requirement of a job is also unknown when it starts processing, and can only be determined by processing the job and observing how long it takes to be completed. In fact, our results show that the traditional sort of on-line scheduling problem is provably not much harder than its off-line analogue, whereas the lack of knowledge about job sizes can drastically affect the quality of solutions that can be obtained.

We shall evaluate on-line algorithms in terms of their *competitive ratio* [116]. Let $C^A_{\max}(\mathcal{I})$ be the makespan of a deterministic on-line algorithm A on instance $\mathcal{I}$. Algorithm A is said to have *competitive ratio c* (or is said to be *c-competitive*) if $C^A_{\max}(\mathcal{I}) \leq c \cdot C^*_{\max}(\mathcal{I}) + O(1)$ for all problem instances $\mathcal{I}$. If A is a randomized algorithm, then A is said have competitive ratio c (or is said to be c-competitive) if $E[C^A_{\max}(\mathcal{I})] \leq c \cdot C^*_{\max}(\mathcal{I}) + O(1)$ for all instances $\mathcal{I}$, where the expectation is taken over all random choices of the algorithm A. Although these notions apply to algorithms without any restrictions on their running times, we will focus on polynomial-time on-line algorithms, rather than the purely information-theoretic analogue. Nonetheless, our lower bounds are based on information-theoretic arguments.

In a non-preemptive model, it may be unrealistic to assume that once a job is started, it must be run until its (unknown) completion time, without any form of recourse. A central aspect of our models is that we introduce the notion of *restarts*: a job may be canceled and later started again, but it is started again from scratch. For example, in the uniformly related machine model, we may wish to cancel a job that is taking longer than "anticipated", and then start it again on a faster machine.

The results of this chapter are as follows. We introduce two general techniques to convert off-line algorithms to algorithms that require less initial information. Using the first technique, we show that we can focus on the case without release dates, since the situation in which there are unknown job arrivals and unknown processing times can be reduced, with only a factor of 2 increase in the competitive ratio, to one in which there are only unknown processing times.

This result also holds when comparing a model in which there are only unknown arrivals and its off-line equivalent. As a consequence, we consider the situation in which all jobs (of unknown size) are available to be scheduled from the start. For both uniformly related and unrelated machines, we use our second technique to convert off-line algorithms into algorithms that need not be given the processing time of each job. Nonetheless, the resulting on-line algorithms do not suffer too great a degradation in the quality of the solution produced.

It is quite simple to obtain tight bounds for the identical machine model: one of the oldest results in scheduling theory is an on-line algorithm of Graham [52], which produces a schedule of length at most $(2 - \frac{1}{m})C^*_{\max}$; we give a straightforward proof that this is exactly the best possible ratio. We also give an identical tight bound on the competitive ratio obtainable in the preemptive variant. This has the important consequence that, although complexity theory shows that there is a fundamental difference between the preemptive and non-preemptive models, this difference disappears when scheduling jobs on-line. We also show that randomization is of little help to the scheduler, proving that no randomized algorithm can achieve competitive ratio better than $(2 - O(\frac{1}{\sqrt{m+1}}))$, even against an oblivious adversary. This result is in sharp contrast to other recent work in on-line algorithms, in which randomness has been shown to significantly increase the performance of the algorithms [71, 119].

We then show that on-line scheduling on uniformly related machines is much harder than on identical machines. This is also quite different from the off-line setting, where results for identical machines have typically extended to the case where machines run at different speeds. In our on-line model, we show that this generalization does make the problem significantly harder: we prove that the optimal competitive ratio is $\Theta(\log m)$. We generalize this model to unrelated machines by assuming that for each job, the relative speeds of the machines are known, but its size is unknown. In this setting, we can also obtain an on-line algorithm with an $O(\log n)$ competitive ratio. Once again, we also give identical results for the preemptive variants of these models. For uniformly related machines, we also show how to take advantage of the situation when the relative speeds of the machines are not too different; we give an $O(\log R)$-competitive algorithm for the non-preemptive model, where R is the ratio of the fastest-to-slowest machine speeds. Finally, we can show that this bound is tight, in the following sense: for any ratio of machine speeds $R < m$ we prove a lower bound of $\Omega(\log R)$ on the competitive ratio of any deterministic on-line scheduling algorithm.

3.1.1 Previous Work on On-line Algorithms

On-line algorithms have been studied for a variety of problem domains. Some of the oldest of these results are for the bin-packing problem, where the quality of an on-line algorithm was similarly measured by the worst-case ratio of the performance of the on-line algorithm to the best off-line algorithm. The idea to use this ratio as a measure of the quality of an on-line algorithm was not utilized in other areas of computer science until the work of Sleator and Tarjan, who studied several questions in paging and list maintenance [115]. These problems are inherently on-line, since one must maintain the list or decide which pages should be in primary memory without knowing what the future sequence of requests will be. Until the work of Sleator and Tarjan, people typically evaluated strategies for these problems by asymptotic average-case analysis. These analyses, however, were not always consistent with experimental evidence about the performance of the various strategies. The idea of using the competitive ratio to evaluate these algorithms proved exciting, since in certain cases their results lent theoretical support to the superiority of the best strategies in practice.

The subsequent growth of work on on-line algorithms was explosive. An attempt to provide a general theory of on-line algorithms was made by Borodin, Linial and Saks, who defined the "metrical task system" and gave tight bounds on the performance of algorithms in this framework [17]. The generality of the characterization led to pessimistic worst-case performance bounds and therefore limited its usefulness.

A less general framework that quickly gained a great deal of notoriety is the k-server problem, introduced by Manasse, McGeoch and Sleator [87]. Given k "servers" in a metric space, the k-server problem requires the service of a sequence of service requests, where a service request is a point in the metric space and is served by moving a server to that point. The problem is on-line in that the future sequence of requests is unknown. This problem generalizes several important problems, including caching, paging and planning the motion of the heads of a two-headed disk.

The famous k-server conjecture is that there is a k-competitive algorithm for this problem; it is known that no better competitive ratio is possible [87]. This conjecture has been proved to be true in a number of special cases [87, 24, 23], and for the general case there exist algorithms whose competitive ratios are known to depend only on k, albeit exponentially [41, 54].

This flurry of work has motivated another stream of work in on-line algorithms, in which people have studied how well one can solve basic problems in combinatorial optimization on-line. Graph coloring [63, 119], matching [71], weighted matching [68], and the transportation problem [72] have all been considered in a model where nodes of the graph (and incident edges) are introduced one at a time and an irrevocable decision must be made about the color of the node or which edge should be in the matching. A variety of path-finding problems have been considered as well, where one tries to find the shortest path possible in an environment about which one does not have total information [94, 29, 15].

The work in this chapter can be seen as in the spi [illegible] ›ot" of these directions of work in on-line algorithms. Scheduling problems are, like paging and caching, basic issues in computer control; they are also basic questions in combinatorial optimization.

3.1.2 Previous Work on On-line Scheduling

A model of on-line scheduling that is very different from ours is closely related to a variant of the bin-packing problem. When the number of bins is fixed, on-line bin-packing can be interpreted as a type of on-line scheduling, where the jobs are given in a list and scheduled in turn. The job currently being scheduled is completely specified, but the jobs later in the list are completely unknown. This model corresponds less to a dynamic environment and more to a first come first serve reservation system for some date in the future. It also bears some similarity to the on-line graph problems mentioned above. Recently Johnson [65] and Chandrasekaran and Narayanan [91] have proved some lower bounds in this model.

In terms of previous work on our model of on-line scheduling, some attention has been given in the past to the question of unknown release dates. In the preemptive model, Gonzales and Johnson gave a polynomial time algorithm that optimally solves this problem on identical machines. Sahni and Cho extended this result to apply to uniformly related machines. In the non-preemptive model Gusfield considered a more general problem on identical machines, in which each job has an associated due date, and the goal is to minimize the maximum lateness. He proved a bound of $(2 - \frac{1}{m})p_{\max}$ on the difference between the maximum lateness produced by an on-line heuristic and the minimum possible maximum lateness.

Relatively little work has been done on the model of on-line scheduling that we consider. Chandra, Karloff, and Vishwanathan [21] proposed studying on-line scheduling with unknown processing times, and analyzed the problem of minimizing the average completion time on a single machine with preemption. In addition to the algorithms for identical machines given by Graham [52], the only other work for parallel machines known to us prior to ours is that of Jaffe [64] and Davis and Jaffe [28]. Davis and Jaffe show that in a restricted model without restarts, an on-line algorithm for non-preemptive scheduling of uniformly related machines cannot have competitive ratio better than $\Omega(\sqrt{m})$. Jaffe gives an algorithm for this case with competitive ratio $O(\sqrt{m})$.

Recently Feldmann, Sgall and Teng [38] studied on-line scheduling on a mesh of identical processors, where one must allocate a submesh of a specified size to a job, when the processing time of that job is unknown. They prove a $\Theta(\sqrt{\log\log m})$ bound on the competitive ratio in this model. In addition, they study a number of other architectures such as hypercubes and trees.

The rest of this chapter is organized as follows. In Section 3.2 we show that the introduction of unknown release dates into a scheduling problem does not make the problem too much harder. As a result we concentrate on the situation where all the jobs are available at time 0 but have unknown processing requirements. In Section 3.3 we present our on-line scheduling algorithms for the various parallel machine models, and in Section 3.4 we give the corresponding lower bounds. In Section 3.5 we discuss on-line scheduling in several other scheduling models, and we suggest some further directions for research in Section 3.6.

3.2 Unknown Release Dates

Our model of on-line scheduling includes both unknown release dates for jobs and unknown job sizes. In this section we will show that, with respect to minimizing schedule length, the first element is much less important than the second element. We will show that if the release dates are unknown, then we can assume that all jobs *are* always available, and repeatedly use an algorithm that works in this environment; the feasible schedules produced by this simulation are of only somewhat lesser quality than can be obtained in the special case. This result does not depend on the remaining specifics of the scheduling environment; in particular, it allows us to use off-line algorithms to obtain algorithms that can handle unknown release dates (but where the processing times are known once released), as well as allowing us to focus on on-line algorithms in the case when all jobs are released at time 0.

Theorem 3.2.1 Let A be a polynomial-time scheduling algorithm which works in an environment in which each job to be scheduled is available at time 0 and always produces a schedule of length at most $\rho C^*_{\max}$. For the analogous environment in which the existence of a job is unknown until its release date, there exists another polynomial-time algorithm A' that works in this more general setting, and produces a schedule of length at most $2\rho C^*_{\max}$.

Proof: Let $\mathcal{I}$ be an instance including jobs with unknown release dates, and let S_0 be the set of jobs available at time 0. The scheduler applies algorithm A and schedules the jobs in S_0, finishing at time F_0. Let S_1 be the set of jobs released in time $(0, F_0]$. The scheduler now, at time F_0, applies algorithm A to schedule S_1, finishing at time F_1. In general let S_{i+1} be the set of jobs released in $(F_{i-1}, F_i]$, and let F_i be the point in time when the schedule for S_i completes. At time F_i, the scheduler uses algorithm A to schedule the jobs in S_{i+1}. Let F_k be the finishing point of the entire schedule. (See Figure 3.1.)

To analyze the length of the resulting schedule, consider the modified problem instance $\mathcal{I}'$ where the jobs in S_k are released at time F_{k-2}. Since these jobs are released at an earlier point in time in $\mathcal{I}'$ than in $\mathcal{I}$, certainly $C^*_{\max}(\mathcal{I}') \leq C^*_{\max}(\mathcal{I})$.

Now note that $F_{k-2} + F_k - F_{k-1} \leq \rho C^*_{\max}(\mathcal{I}')$, since the jobs in S_k are not released until F_{k-2} and the properties of algorithm A guarantee that $F_k - F_{k-1}$ is within a factor of ρ of the shortest schedule for S_k. Similarly, $F_{k-1} - F_{k-2} \leq \rho C^*_{\max}(\mathcal{I})$. Therefore $F_k \leq 2\rho C^*_{\max}(\mathcal{I}') \leq 2\rho C^*_{\max}(\mathcal{I})$. ∎

This theorem is very general, in that it can be applied to a number of different types of scheduling scenarios. In particular, it shows that to produce an on-line algorithm for our full on-line model, we can modify an algorithm for the case in which all jobs are available at time 0 and processing times are unknown, increasing the competitive ratio of the algorithm by only a factor of 2. Further, the theorem applies not only to problems of parallel machine scheduling but also to the entire class of shop scheduling problems, including open shop, flow shop and job

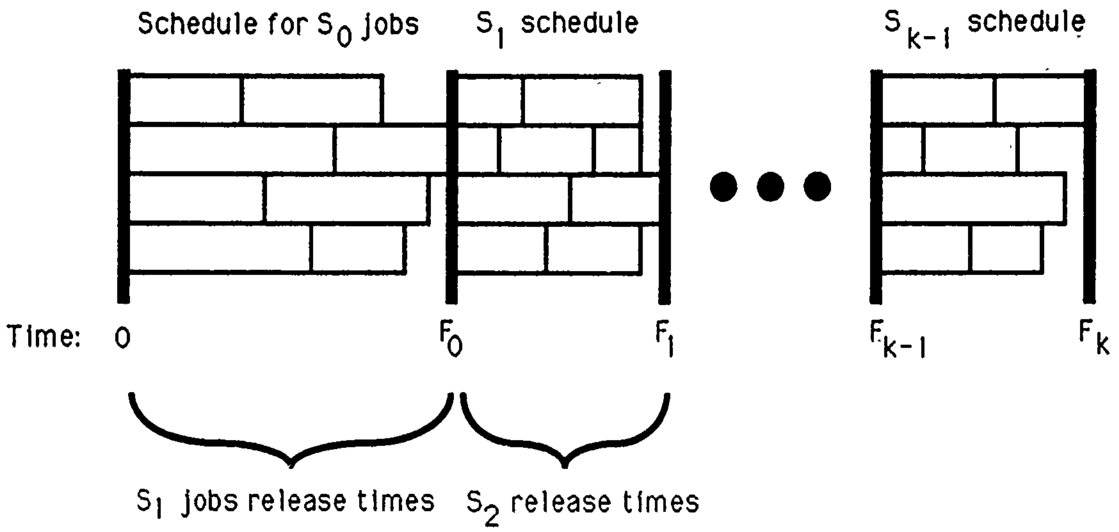

Figure 3.1: Using an algorithm for a scheduling environment without release dates to schedule in an environment with release dates.

shop [112]. In addition it applies to the scheduling model of Feldman, Sgall and Teng mentioned in section 3.1.2. They studied the on-line allocation of submeshes of a large mesh to different jobs, but their algorithms only worked when all jobs were available at time 0. Our theorem generalizes their result to a $\Theta(\sqrt{\log\log m})$ on-line algorithm even when jobs have unknown release dates. Finally, our theorem yields the following corollary:

Corollary 3.2.2 If job release dates are unknown, but once a job arrives its size is known, there is a polynomial-time *on-line* algorithm for scheduling uniformly related machines that comes within a factor of $(2+\epsilon)$ of optimal and a polynomial-time *on-line* algorithm for scheduling unrelated machines that comes within a factor of 4 of optimal.

Proof: Direct from the theorem and previously known results on the approximability of these problems [61, 83].

For identical machines this result yields a $(2+\epsilon)$-approximation algorithm; however, something slightly better was already known. In 1966 Graham showed that list scheduling was a $(2-\frac{1}{m})$-approximation algorithm for scheduling identical machines. In list scheduling, the scheduler takes any list of jobs and, whenever a machine becomes available, places the next job on the list on that machine. It is not hard to see that if this strategy is extended so that newly arriving jobs are added to the end of the list, then list scheduling is a $(2 - \frac{1}{m})$-approximation algorithm for scheduling identical machines with release dates. See, for example, [56].

Despite the fact that unknown release dates do not make a scheduling problem *much* more difficult, we can show that they sometimes do make it more difficult to schedule machines near-optimally.

Theorem 3.2.3 There is no on-line algorithm for non-preemptive scheduling of identical machines with unknown release dates but known processing requirements with competitive ratio better than 10/9, even if restarts are allowed.

It is interesting to note that this is not the case when preemption is allowed; Sahni and Cho have shown that there is a polynomial-time algorithm to solve that problem optimally [103]. Intuitively this is not surprising, since preemption allows you to adjust to new information without losing work done beforehand.

Proof: Consider the machine environment that consists of two identical machines. At time 0 there are two jobs of size 3 (A and B) released and one job of size 2 (C). We consider several cases.

- The scheduler initially schedules A on machine 1 and B on machine 2 at time 0, and restarts neither until (possibly) time 2. In this case the adversary introduces a job (D) of size 4 at time 2. If the scheduler does not interrupt A or B and start D at time 2 the minimum schedule length will be 7; if the scheduler does interrupt one to run job D then the minimum schedule is of length 8. The optimal schedule would have been of length 6: C and D on machine 1 and A and B on machine 2. In this case the performance ratio is at least $\frac{7}{6}$.

- The scheduler initially schedules A on machine 1 and B on machine 2, and interrupts at least one of them at time 1. At time 2 the adversary releases a job of size 2; the optimal schedule for this example is of length 5, but we've forced a schedule of length 6, therefore the performance ratio is at least $\frac{6}{5}$.

- The scheduler only schedules one job to begin at time 0. In this case the adversary introduces a job of size 2 at time 1; the resulting performance ratio is at least $\frac{6}{5}$.
- The scheduler schedules A and C at time 0. The adversary then introduces a job of size 7 at time 1. If the scheduler does not interrupt A or C at time 1, the minimum length of the produced schedule is 9, whereas the optimal schedule would be of length 8; hence the performance bound is at least $\frac{9}{8}$. If the scheduler preempts A or C at time 1, the introduces a job of size 3 at time 3. The optimal schedule for this instance is of length 9, and the algorithm produces a schedule of length at least 10. Therefore the performance bound in this case, and in general, is at least $\frac{10}{9}$. ■

In light of the results in this section, for the remainder of this chapter we shall focus on scheduling environments in which all jobs are available to be scheduled at time 0.

3.3 Algorithms for On-Line Scheduling

In this section we will present on-line scheduling algorithms for the basic parallel machine models. We first note that in the case of identical machines, the well-known list scheduling algorithm of Graham [52] always comes within a factor of $(2 - \frac{1}{m})$ of the optimal length schedule, and comes within the same bound of the optimal preemptive schedule length. Since list scheduling does not depend on the sizes of the jobs, list scheduling is an on-line algorithm with a $(2 - \frac{1}{m})$ competitive ratio.

Theorem 3.3.1 [Graham] There is an on-line algorithm for scheduling identical machines that achieves competitive ratio $(2 - \frac{1}{m})$ in both the preemptive and non-preemptive models.

For the other machine models, we will present a general technique which yields an $O(\log n)$-competitive algorithm for each of them. We will then show how to convert this general algorithm to an $O(\log m)$-competitive algorithm for both preemptive and non-preemptive uniformly related machines. We will also present an algorithm for non-preemptive uniformly related machines which has a competitive ratio of $O(\min(\log m, \log(s_1/s_m))$, assuming $s_1 \geq s_2 \geq \cdots \geq s_m$.

3.3.1 The General Technique

Our general algorithm depends on the existence of either polynomial-time algorithms or polynomial-time *ρ-approximation algorithms* for scheduling in the various machine models.

Theorem 3.3.2 **Suppose that there is a ρ-approximation algorithm A for the [non-preemptive/preempt [uniformly related/unrelated] machine problem, and let $\mathcal{I}$ be an instance of this problem. Then there is an on-line scheduling algorithm which produces a schedule no longer than** $(4\rho \log n + 4\rho \log 2\rho + 1)C^*_{\max}(\mathcal{I})$.

Proof: The on-line algorithm works by repeatedly applying algorithm A to the jobs, after guessing the size of each job. Given a schedule produced by the algorithm A, our on-line algorithm will run each job at the particular time interval and on the particular machine specified by the schedule. In the preemptive model, the job may not have finished all its processing by the end of the schedule, in which case we preempt the job. In the non-preemptive model, we cancel the job if it is not completely processed in the time allotted. In either case, if the job does not complete we will be able to update our estimate of the size of that job.

For the sake of simplicity, we will assume that the data is normalized so that the fastest machine for each job J_j has speed $s_{ij} = 1$. One result of this assumption is that any job of size p_j takes time p_j to complete on the machine that processes it fastest.

The complete on-line algorithm is below.

Step 1 Pick any job J'_j and run it to completion on a machine $m_{i'}$ such that $s_{i',j'} = 1$. Let the time that this takes be denoted by Δ.

Step 2 Let $q = \Delta/\rho n$.

Step 3 Use algorithm $\mathcal{A}$ to construct a schedule for all jobs that have not yet completed, setting $p_j \leftarrow q$ for all remaining jobs J_j. Run the jobs in this schedule, preempting or canceling all jobs that do not complete in the time allotted to them by the schedule.

Step 4 If any jobs have not yet completed, set $q \leftarrow 2q$ and go to Step 3.

Let $C^*_{\max}$ be the length of the optimal schedule. We will now analyze the algorithm and the length of the schedule it produces. First, in Step 1, the time Δ taken by job $J_{j'}$ on machine

$m_{i'}$ is at most $C^*_{\max}$, since the optimal schedule can be no shorter than the time taken by any job running on the machine which processes it the fastest.

Next, we show that the first iteration of Step 3 produces a schedule no longer than $\Delta \leq C^*_{\max}$. One way to construct a schedule is to assign each of the n jobs to the machine that processes it the fastest. In the worst case, all n jobs would be assigned to the same machine, and this schedule would have length $nq = \Delta/\rho$. Since the schedule produced by algorithm A is no longer than ρ times optimal, it must produce a schedule of length no longer than $\Delta \leq C^*_{\max}$.

In addition, future iterations of Step 3 must produce schedules of length no longer than $2\rho C^*_{\max}$. Suppose the algorithm performs an iteration of Step 3 in which the jobs are assigned size q. Since the algorithm did not finish processing all jobs in the previous iteration, we know that the instance being scheduled must have some jobs J_j such that $p'_j > q/2$. An optimal schedule for this subset of jobs must take time no greater than $C^*_{\max}$. Ensuring that each of these jobs gets processed for q units can increase the length of the optimal schedule for these jobs by no more than a factor of 2. Finally, the algorithm A will find a schedule for these jobs that is no more than ρ times as long as the optimal schedule, so that the schedule can be no longer than $2\rho C^*_{\max}$.

To derive our $O((\log n)C^*_{\max})$ bound on the length of the schedule, we show that we essentially need to consider only the last $\log(2\rho n)$ iterations of Step 3.

Lemma 3.3.3 Suppose there are f iterations of Step 3. Then the length of the schedule produced in iteration $f - i$ is at least 2^ℓ times as long as the length of the schedule produced in iteration $f - i - \ell\log(2\rho n)$.

Proof: Assume that the (estimated) job size in iteration $f-i$ is q; then the (estimated) job size in iteration $f-i-\ell\log(2\rho n)$ is $q/(2\rho n)^\ell$. If a job with size q exists, then the schedule containing it must take time at least q. As we showed earlier, a schedule produced by algorithm A for jobs with size $q/(2\rho n)^\ell$ has length at most $\rho n q/(2\rho n)^\ell$. Thus the length of the schedule produced when the job size is $q/(2\rho n)^\ell$ is at most $(1/2)^\ell$ times the length of the schedule produced when the job size is q. ■

Since every $\log(2\rho n)$ iterations the length of the schedule produced doubles, we can "charge" iterations $f - i - \ell\log(2\rho n)$, $1 \leq \ell \leq (f-i)/\log(2\rho n)$, to iteration $f - i$. Since each of the last $\log(2\rho n)$ iterations has length no longer than $2kC^*_{\max}$, and each gets charged no more than $\frac{1}{2}+\frac{1}{4}+\cdots+(\frac{1}{2})^{\lceil \frac{l}{\log 2\rho n} \rceil}$ times its length, the overall length of the schedule is at most

$$\Delta + (\log 2\rho n)(2\rho C^*_{\max}(1+\frac{1}{2}+\cdots+(\frac{1}{2})^{\lceil \frac{l}{\log 2\rho n} \rceil})) \leq 4\rho C^*_{\max}\log n + 4\rho \log 2kC^*_{\max} + C^*_{\max},$$

which is $O((\log n)C^*_{\max})$. ■

Since there exists a polynomial-time algorithm for the optimal scheduling of preemptive unrelated machines due to Lawler and Labetoulle [79], and there exists a 2-approximation algorithm for scheduling non-preemptive unrelated machines due to Lenstra, Shmoys, and Tardos [83], we have the following corollaries.

Corollary 3.3.4 There is an on-line algorithm for scheduling preemptive unrelated machines that has competitive ratio $4(\log n) + 5$.

Corollary 3.3.5 There is an on-line algorithm for scheduling non-preemptive unrelated machines that has competitive ratio $8(\log n) + 17$.

We can do somewhat better with uniformly related machines. As the following lemma shows, by applying a list scheduling algorithm until there are at most m unfinished jobs we can quite easily reduce the number of jobs from n to m.

Lemma 3.3.6 The number of jobs in any uniformly related machine problem instance can be reduced on-line from n to m, while increasing the competitive ratio by 1.

Proof: We simply place jobs on machines arbitrarily. Whenever a job completes and a machine falls idle, we assign it a new, unprocessed job. When no new jobs are available, at most m jobs have not yet finished processing. Furthermore, the length of the schedule to this point in time can be no greater than $\sum_j p_j / \sum_i s_i$, which is a lower bound on the length of the optimal preemptive and non-preemptive schedules. ■

Since there is a polynomial-time algorithm for preemptive uniformly related machines due to Horvath, Lam, and Sethi [62], and also a $(1+\epsilon)$-approxmation algorithm for non-preemptive uniformly related machines due to Hochbaum and Shmoys [61], the preceding theorem and lemma yield the following corollaries.

Corollary 3.3.7 There is an on-line algorithm for scheduling preemptive uniformly related machines that has competitive ratio $4(\log m) + 6$.

Corollary 3.3.8 There is an on-line algorithm for scheduling non-preemptive uniformly related machines that has competitive ratio $(4 + \epsilon)(\log m) + (4 + \epsilon)\log(2 + \epsilon) + 2$.

How many preemptions/restarts does this general algorithm perform? In each iteration it is possible that no job finished and therefore there are n preemptions/restarts at the end of the iteration. If $p_{\min}$ is the minimum job size and $r = p_{\max}/p_{\min}$, the on-line algorithm does $O(\log(r\rho n))$ iterations. This is because the Δ established in step 1 can be no smaller than $p_{\min}$; we then set $q = \Delta/\rho n$ and successively double it until we reach $p_{\max}$. Therefore the algorithms for unrelated machines do $O(n \log(n\rho r))$ preemptions/restarts, and those for uniformly related machines do $O(m \log(m\rho r))$.

3.3.2 An Improved Algorithm for Non-Preemptive Uniformly Related Machines

In the case of non-preemptive uniformly related machines, we can obtain an even better bound when the ratio between the speeds of the fastest and slowest machines is less than m. Let $R = s_1/s_m$. We will give an algorithm with competitive ratio $O(\min(\log R, \log m))$. This algorithm uses a new and simple off-line 2-approximation algorithm for uniformly related machines.

A Simple (Off-Line) 2-Relaxed Decision Procedure for Uniformly Related Machines

First we give a new (off-line) 2-relaxed decision procedure for uniformly related machines that will be the basis of our on-line algorithm. The notion of a *ρ-relaxed decision procedure* was used by Hochbaum and Shmoys [60]: given a deadline d, such a procedure either produces a schedule of length ρd or verifies that there exists no schedule of length d. By using binary search, a ρ-relaxed decision procedure can be converted into a ρ-approximation algorithm.

The 2-relaxed decision procedure is as follows. Each machine has an associated queue. Each job is placed into the queue of the slowest machine m_k such that $p_j \leq s_k d$; that is, the slowest machine that can complete the job within the given deadline. If for some job there is no such machine it is clear that there does not exist a schedule of length d. To construct a schedule, whenever a machine is idle, it starts processing a new, unprocessed job from its queue. If a machine's queue is empty, it takes a job to process from the queue of the fastest machine that is slower than it and that has a nonempty queue. If all such queues are empty, then the machine remains idle. If the schedule constructed has $C_{\max} > 2d$, output **no**. Otherwise we have produced a schedule of length at most $2d$.

In order to prove that this is a 2-relaxed decision procedure, we must prove that when the procedure outputs **no** there is no schedule of length d. Consider a job j that was not finished by time $2d$. Since jobs are only processed by machines on which they take less than d units of time, this job must have started after time d; thus it was on the queue of some machine m_k until time d. This implies that until time d machines $m_1, \ldots, m_k$ were all busy processing jobs that could not have completed on machines $m_{k+1}, \ldots, m_m$. Therefore in a schedule of length d it is impossible to process all of these jobs and job j, and so there exists no schedule of length d.

The On-line Algorithm for Non-preemptive Uniformly Related Machines

In this section we will first give an $O(\log R)$-competitive on-line algorithm for non-preemptive uniformly related machines. We will then prove that any problem instance can be reduced, on-line, to an instance where $R \leq m$ while only causing a slight increase in the competitive ratio of an on-line algorithm on that instance. Hence we will have an on-line algorithm for all instances with competitive ratio $O(\min(\log R, \log m))$.

First we present the main algorithm.

Theorem 3.3.9 Let $\mathcal{I}$ be an instance of the scheduling problem for non-preemptive uniformly related machines. Then there is an on-line scheduling algorithm which produces a schedule no longer than $16(\log R)C^*_{\max}(\mathcal{I})$.

Proof: We round machine speeds down to the nearest power of two: when a machine finishes processing a job it pretends to keep processing it long enough so that it seems to have been processed at the lesser speed. When we interpret the schedule for this rounded problem instance as a schedule for the actual problem instance, the competitive ratio can be increased by at most a factor of two. Since the s_i are all powers of two, and all the s_i are within a factor of R of s_1, it immediately follows that there are at most $\log R$ different machine speeds. Let $M_1 = \{m_i | s_i = s_1\}$, $M_2 = \{m_i | s_i = s_1/2\}, \ldots, M_{\log R} = \{m_i | s_i = s_1/2^{\log R}\}$.

Our strategy will be to first convert the off-line 2-relaxed decision procedure into an on-line $2\log R$-relaxed decision procedure, and then build from that an on-line algorithm. The off-line decision procedure does not immediately lend itself to an on-line procedure, since the criterion it uses to assign jobs to machine queues utilizes knowledge of the job sizes. To convert this to an on-line decision procedure we will repeatedly run the off-line relaxed decision procedure to either schedule a job or else update the estimate of its size. Note that given the rounded machine speeds, instead of queueing jobs on machines $m_1, \ldots, m_m$, we can instead queue jobs on sets of machines $M_1, \ldots, M_{\log R}$.

A formal description of an on-line $2\log R$-relaxed decision procedure is as follows. The procedure either outputs **no** if there is no schedule of length d or it produces a schedule of length $2d\log R$. Note that even if it answers **no** the procedure may have completely processed some of the jobs in that time.

Input A set of jobs and a deadline d.

Step 0 Put all jobs into the $M_{\log R}$ queue.

Step 1 Run the off-line 2-relaxed decision procedure, with the modification that no jobs are started after time d (that is, when a machine is idle it takes a job to process off of its queue, or, when its queue is empty, off of the first slower machine that has a non-empty queue; etc.)

Step 2
1. **If** all jobs finish processing by time $2d$ **stop**.
2. **If** any machine in M_1 is still processing a job at time $2d$ then there is no schedule of length d. Output **no**; **return**.
3. **If** any set of machines M_k has a job j in its queue at time d then there is no schedule of length d. Output **no**; **return**.
4. **If** there are jobs that are being processed at time $2d$, on machines in M_i, $i > 1$, cancel these jobs and put them on the queue of M_{i-1}. Go to Step 1.

To prove that this is an on-line $2\log R$-relaxed decision procedure notice first that the length of the schedule or partial schedule produced by this procedure is no longer than $2d\log R$, since the off-line relaxed decision procedure produces a schedule of length at most $2d$ and it is run at most $\log R$ times. Furthermore, despite the fact that the p_j are unknown, the on-line relaxed decision procedure maintains the invariant that a job is only on the queue of M_k if it could not complete in time d on any machine in $M_{k+1}, \ldots, M_{\log R}$. This is certainly true initially, since all the jobs are put in the queue of $M_{\log R}$. Furthermore, since the procedure does not start new jobs after time d, any job that is still being processed at time $2d$ on some machine in M_i must take more than time d to process on any machine in M_i. Therefore any such job does not belong on the queue of M_i or any slower set of machines, and is placed on the queue of M_{i-1}.

Now we will show that if the procedure outputs **no** then there is no schedule of length d. If condition 2 is true then a machine in M_1 ran a job for more than d units of time; therefore this job could not have been processed in d units of time on any of the machines, since no other machine runs at a faster speed. If condition 3 is true, then up until time d all machines in the sets $M_1, \ldots M_k$ must have been busy processing jobs that could not have been processed in d units of time on machines in $M_{k+1}, \ldots, M_{\log R}$. Therefore, machines in $M_1, \ldots, M_k$ could not have processed all of these jobs and job j as well by time d.

We have given an on-line $2\log R$-relaxed decision procedure; we now show how to use it to develop an on-line $O(\log R)$-approximation algorithm.

The on-line algorithm initially establishes a lower bound Δ on $C^*_{\max}$ by running an arbitrarily chosen job on the fastest processor. Let Δ be the time taken to complete this job; certainly $\Delta \leq C^*_{\max}$. Next, the on-line algorithm calls the procedure on the set of all jobs with $d = \Delta$. If the procedure returns **no**, then we will call it again with $d = 2\Delta$ and the set of jobs that were not completely processed in the first iteration. In general, if the ith iteration fails to produce a schedule, then we will call the procedure again for the $(i+1)$st time with $d = 2^i\Delta$ and all jobs that have not yet been completely processed. Observe that if the ith iteration fails to produce a schedule when called with $d = 2^{i-1}\Delta$, then it proves that $2^{i-1}\Delta < C^*_{\max}$. Suppose that we finally finish processing all jobs in iteration f. Then the total length of the schedule produced is

$$2 \cdot (\Delta + (1 + 2 + \cdots + 2^{f-1})(2\Delta \log R)) \leq 2^{f+2}\Delta \log R.$$

Since the procedure failed to produce a schedule in iteration $f-1$, we know that $2^{f-2}\Delta < C^*_{\max}$. Therefore the total length of the schedule produced is no greater than $16(\log R)C^*_{\max}$. ■

If we consider only the k fastest machines, where k is defined as the smallest k such that such that $\sum_{i=1}^{k} s_i \geq \frac{1}{2}\sum_{i=1}^{m} s_i$, then $R = s_1/s_k \leq m$. We now show that we can assume that that the above algorithm uses only the k fastest machines. In time $2C^*_{\max}$ we can process on-line all but k of the jobs by processing jobs arbitrarily on machines $m_1, \ldots, m_k$ until the first moment in time at which at most k jobs have not yet been completely processed. The amount of time it takes until this point is bounded above by $(\sum_{j=1}^{n} p_j)/(\frac{1}{2}\sum_{i=1}^{m} s_i) \leq 2C^*_{\max}$, since none of the k machines is idle. We will only need machines $m_1, \ldots, m_k$ to process these last k jobs. If we then produce a schedule of length l for the last k jobs on these machines, the entire schedule will be of length $2C^*_{\max} + l$. Therefore, without loss of generality, we can assume that the machine speeds satisfy $R \leq m$.

Corollary 3.3.10 There is an on-line algorithm for scheduling non-preemptive uniformly related machines that has competitive ratio $\min(16 \log R, 16 \log m + 2)$.

To bound the number of restarts of this on-line algorithm, observe that in any iteration of the on-line relaxed decision procedure at most $O(m)$ jobs will be restarted $O(\log R)$ times. The on-line relaxed decision procedure is run at most $O(\log(C^*_{\max}s_1/p_{\min}))$ times, since the initial candidate deadline is at least $p_{\min}/s_1$ and we successively double the deadline until we reach a feasible deadline, which $C^*_{\max}$ certainly is. Therefore this algorithm performs $O(m \log R \log(C^*_{\max}s_1/p_{\min}))$ restarts.

3.4 Lower Bounds

3.4.1 Identical Machines

As with other on-line algorithms, on-line scheduling can be viewed as a game against an adversary who is allowed to determine the information that is revealed incrementally to the algorithm.

Therefore, our lower bound arguments will often be phrased in terms of a strategy for an adversary, who attempts to reveal information in such a way as to force the competitive ratio to be as large as possible. We will consider two possible types of adversaries. The stronger is an *adaptive adversary*, who knows in advance the scheduling algorithm and also knows in advance the result of any coin tosses of the algorithm. The weaker is an *oblivious adversary*, who knows only the algorithm but not the coin tosses [100, 7]. Note that for deterministic algorithms this distinction is clearly irrelevant.

The oblivious adversary models the situation where a randomized algorithm receives a problem instance and must produce a solution; the random choices it makes have no effect on the input it sees. On the other hand, the adaptive adversary models the situation where there is some feedback between the choices the algorithm makes and future input it sees. A good example of this latter situation is paging: depending on what random choices a paging algorithm makes it may or may not itself cause page faults, in addition to those caused by other elements of the operating system.

We begin our lower bounds with a lower bound on the competitive ratio of any on-line algorithm for scheduling identical machines.

Theorem 3.4.1 The competitive ratio of any deterministic on-line algorithm for scheduling identical machines, with no preemption allowed, is at least $(2 - \frac{1}{m})$.

Proof: For any m, let $n = m(m-1)+1$. Each of the first $m(m-1)$ jobs is of size 1, while the last job is of size m; that is, $p_1 = \cdots = p_{n-1} = 1, p_n = m$. This instance is due to Graham [52]. The optimal schedule is of length m, and consists of scheduling the last job on a machine by itself, and scheduling m of the single unit jobs on each of the remaining $m-1$ machines. The length of a schedule for this instance is determined by the starting time of the job of size m; therefore the adversary wishes to make it start as late as possible. Each of the first $n-1$ jobs that the scheduler allows to run for at least one unit of time will be fixed by the adversary to be jobs of size 1. Given this strategy of the adversary, it is not difficult to see that by time i, $1 \leq i \leq m-1$, at most im jobs are either completely processed or currently being processed. Hence by time $m-1$ there must be one job that has not been completely processed and is not currently being processed. The adversary sets this job to be of size m. If this job starts at time

$m-1$ the fastest the schedule can complete is by time $2m-1$, which is $(2-\frac{1}{m})$ times as long as the optimal schedule. ∎

In contrast to the nonpreemptive model an optimal preemptive schedule can be found off-line in polynomial time [89]. Interestingly enough, an argument similar to the previous proof shows that the on-line worst-case characterization of both models is the same.

Theorem 3.4.2 **Any deterministic on-line algorithm for scheduling identical machines with preemption allowed has competitive ratio at least $(2-\frac{1}{m})$.**

Proof: Consider an instance with $n = m+1$ jobs. The adversary allows the scheduler to begin scheduling, and waits until either the scheduler preempts a job for the first time, or 1 time unit has passed, whichever comes first. Call this time t. By time t, at most m jobs can have been started (since scheduler didn't preempt anything until time t). Let job n be a job that was not started. At time t, the adversary sets $p_1 = \cdots = p_{n-1} = t$, and sets $p_n = tm/(m-1)$. The scheduler can clearly complete the entire schedule no sooner than time $t + tm/(m-1)$. The length of the optimal preemptive schedule is known to be $\max(p_{\max}, \sum p_j/m)$. In this case $\max(p_{\max}, \sum p_j/m) = tm/(m-1)$. Therefore the adversary has competitive ratio $[t+tm/(m-1)]/[tm/(m-1)] = (2-\frac{1}{m})$. ∎

The essence of these deterministic lower bounds is that there is one large job whose starting time determines the length of the schedule, and the adversary can force the scheduler to start that job late in the schedule. When we move to randomized algorithms it is true that an *adaptive* adversary can force the randomized scheduler to do as poorly as the deterministic scheduler. One might imagine, however, that a randomized algorithm A working against an *oblivious* adversary might, with significant probability, select and schedule the large jobs earlier, thus doing better. (It is known, for example, that an algorithm that schedules jobs in nonincreasing size order produces a schedule of length no longer than $\frac{4}{3}C^*_{\max}$ [53].) A randomized algorithm can clearly gain *something* over a deterministic algorithm: consider, for example the algorithm that randomly chooses an ordering of the jobs and then list schedules according to that ordering. Since each list schedule is at most $(2-\frac{1}{m})$ times optimal in length, and at least one of the list schedules will be the optimal schedule, this randomized algorithm has expected performance strictly less than $(2-\frac{1}{m})$. We will prove, however, that randomness is ultimately of little help to a non-preemptive on-line scheduler of identical machines.

Theorem 3.4.3 Any randomized on-line algorithm for non-preemptive scheduling of identical machines, working against an oblivious adversary, has worst case expected value of at least $(2 - O(\frac{1}{\sqrt{m+1}}))C^*_{\max}$.

Our strategy to prove this theorem is as follows. We will first define the notion of a *reasonable* randomized algorithm for scheduling identical machines. We will then show that for any c-competitive unreasonable algorithm, there exists a reasonable algorithm that has a competitive ratio no greater than c and that always chooses the next job to schedule uniformly. Finally, we will provide an instance for which the competitive ratio of such a strategy has worst case expected value $(2 - O(\frac{1}{\sqrt{m+1}}))C^*_{\max}$.

Definition 3.4.4 A *reasonable* randomized algorithm for scheduling identical machines is an algorithm that does not restart any job and does not leave any machine idle so long as there is some job that has not yet been started.

Lemma 3.4.5 For any unreasonable algorithm A there is a reasonable algorithm A' whose worst-case expected performance is at least as good as that of A.

Proof: First we argue that the introduction of idle time into a schedule cannot help the scheduler. Assume that job j is to be started at time t_2 on machine i which is idle from time t_1 to t_2. Now if job j is available at time t_1, it is clearly to the advantage of the scheduler to start job j on machine i at time t_1. If job j is not available at time t_1 then it is running on another machine i'. In this case there is no point in restarting job j on machine i; since the two machines are of identical speed we can switch the future schedules of the two machines without increasing the total length of the schedule. Now restarting a job j after it has run for $t < p_j$ units of time is equivalent, in terms of the effect on the length of the schedule, to introducing t units of idle time, and thus does not help the scheduler either.

Lemma 3.4.6 A reasonable randomized algorithm A is equivalent to an algorithm that, whenever a machine becomes idle, picks one of the unstarted jobs with a certain probability distribution which may depend on the schedule constructed up to that point.

Proof: Since a reasonable randomized algorithm constructs a schedule with no restarts and no idle time, it must be the case that it schedules some unstarted job whenever a machine becomes idle. The probability distribution for its next choice cannot depend on information that the algorithm does not have at that point; thus, it can depend only on the schedule constructed until that particular choice of a job. ■

We will now argue that the adversary can always force the scheduler to do as poorly as it would have done had it always made its choices according to the uniform distribution.

Lemma 3.4.7 The competitive ratio of a reasonable randomized algorithm A can be no less than that of the reasonable algorithm U that always picks the next job to process uniformly from among the remaining jobs.

Proof: We note that the adversary's strategy can be described as choosing the sizes of the jobs and then choosing some permutation of the jobs. If the adversary chooses the permutation randomly and uniformly, then the probability of the algorithm A selecting any particular job is uniform over all remaining jobs, no matter what probability distribution A uses. Let $\mathcal{E}$ be the expected performance of algorithm A against this adversary, where the expectation is taken over the random choices of both A and the adversary. Note that the adversary can always choose some permutation of jobs such that the expected performance of A, taken over just the choices of A, is no better than $\mathcal{E}$. Since the expected performance of the algorithm U that chooses uniformly is $\mathcal{E}$ no matter which permutation is used, algorithm A can have competitive ratio no better than algorithm U. ■

We complete the proof of theorem 3.4.3 by showing that scheduling by choosing the next job uniformly can do quite poorly.

Lemma 3.4.8 There is a problem instance for scheduling identical machines on which a uniform choice of the next job to process produces a schedule with expected length $(2 - O(\frac{1}{\sqrt{m+1}}))C^*_{\max}$.

Proof: We will consider the problem instance with k jobs of size m ("big" jobs) and $m(m-k)$ jobs of size 1 ("small" jobs). The optimum length schedule for this instance is of length m. The expected length of the schedule is then $m + E_s$, where E_s is the expected start time of the last big job in the schedule. To bound E_s, we can think of the problem as a "shell game", where there are $n = m(m-k)+k$ shells, under k of which there are peas. If one searches for the peas by choosing among the remaining shells randomly and uniformly, the expected place of the last pea to be found is $\frac{k}{k+1}(n)$. Therefore we expect the kth largest job to be the $\frac{k}{k+1}[m(m-k)+k]$th chosen overall. This will happen no earlier than time $\frac{k}{k+1}(m-k)$, since at most m jobs are completed during every unit of time. Therefore $E_s \geq \frac{k}{k+1}(m-k) = F(k)$. To derive the strongest lower bound possible we maximize $F(k)$ by setting the first derivative to 0.

$$F'(k) = \frac{(m-2k)(k+1)-(k(m-k))}{(k+1)^2},$$

so we wish to solve

$$(m-2k)(k+1)-(k(m-k)) = -k^2-2k+m = 0.$$

This implies that $k = \sqrt{m+1}-1$; plugging into $F(k)$ we see that

$$\begin{aligned} F(k) &= \frac{\sqrt{m+1}-1}{\sqrt{m+1}}\left(m-\sqrt{m+1}+1\right) \\ &= \frac{m(\sqrt{m+1})-(m+1)+\sqrt{m+1}-m+\sqrt{m+1}-1}{\sqrt{m+1}} \\ &= \frac{m\sqrt{m+1}+2\sqrt{m+1}-2m-2}{\sqrt{m+1}} \\ &= m\left(1-\frac{2}{\sqrt{m+1}}+\frac{2}{m}-\frac{2}{m(\sqrt{m+1})}\right) \end{aligned}$$

Thus $m+E_s = m(2-\frac{2}{\sqrt{m+1}}+o(\frac{1}{\sqrt{m+1}})) = (2-\frac{2}{\sqrt{m+1}}+o(\frac{1}{\sqrt{m+1}}))C^*_{\max}$ which implies the stated result. ■

3.4.2 Uniformly Related Machines

In the case of uniformly related machines the situation becomes significantly more difficult for the scheduler. We will show that the adversary can force any deterministic scheduler to construct a schedule of length $\Omega(\log m)$ times the length of the optimal schedule, whether or not the scheduler is allowed to preempt jobs.

Theorem 3.4.9 The competitive ratio of any deterministic on-line scheduling algorithm for uniformly related machines, whether or not preemption is allowed, is $\Omega(\log m)$.

Our proof will be in terms of a preemptive scheduler, but it is not hard to see that a similar argument will work when no preemption is allowed.

To prove this theorem we use a family of instances $\mathcal{I}_k$ for uniformly related machines given by Cho and Sahni [22] in a somewhat different context. Let $k = (\log_2(3m-1)+1)/2$. We restrict ourselves to values of m such that k is integral. The instance $\mathcal{I}_k$ has k sets of machines G_i and k sets of jobs T_i, $1 \leq i \leq k$. Each machine in G_i has a speed of 2^i and each job in T_i has size 2^i. Finally, $|G_i| = |T_i| = 2^{2k-2i-1}$ for $1 \leq i < k$, and $|G_k| = |T_k| = 1$. It is easy to see that $C^*_{\max} = 1$, since each job of size p_j can be scheduled by itself on a machine of speed p_j.

Let X_i be the time when, in a schedule for $\mathcal{I}_k$, the last job in T_i finishes.

Lemma 3.4.10 The adversary can always force the scheduler to construct a schedule for $\mathcal{I}_k$ in which $X_1 \leq X_2 \leq \ldots \leq X_k$.

Proof: Assume that the adversary is competing against a scheduler who somehow knows the job sizes in advance, but doesn't know which size belongs to which job. Certainly if the adversary can force this type of scheduler to do badly, the adversary can force a scheduler with no knowledge of job sizes to do badly. We introduce the idea of the adversary *committing* to a set of jobs. At time t, let $J(t)$ be the set of jobs that have not yet completed. The scheduler has a corresponding set $L(t)$ of the sizes of the jobs which have not yet completed. The adversary is not committed to any job in $J(t)$ if, given the amount of time that the jobs in $J(t)$ have been running, the scheduler cannot infer any information about which job in $J(t)$ is associated with which size in $L(t)$. More formally, the adversary is *not committed to any job in* $J(t)$ if, at time t, any bijective mapping from $J(t)$ to $L(t)$ is valid given the schedule thus far. Let $R(i,t)$ be the total amount of processing that has been done on the job that is running on machine i at time t. If the adversary is not committed to any job in $J(t)$ at time t then

$$R(i,t) \leq \min_{j \in J(t)} p_j, \; 1 \leq i \leq m.$$

The adversary's strategy is to avoid being committed to any job in $J(t)$. The adversary can do this, if, at any point in time t' such that $R(i,t') = \min_{j \in J(t')} p_j$ for some i, the adversary allows the smallest job in $J(t')$ to complete on machine i. If the equality holds true for more than one machine i or more than one job j, then the smallest indexed job j completes on the smallest indexed machine i and so forth. The adversary continues to complete jobs until the inequality $R(i,t') < \min_{j \in J(t')} p_j$ holds again. It is clear that this strategy yields a schedule satisfying the condition of the lemma. ■

Lemma 3.4.11 The adversary can always force the scheduler to produce a schedule for $\mathcal{I}_k$ in which $X_i - X_{i-1} \geq \frac{1}{4}$, $1 \leq i \leq k$, $X_0 = 0$.

Proof: The adversary uses the same strategy as in the previous proof. Consider the status of the jobs in T_{i+1} at time X_i. None of them have been completed; in fact, no more than 2^i of the 2^{i+1} units of each job have been processed. This is because until all the jobs in T_i were completed (at time X_i), any job that had 2^i units processed was designated by the adversary to be in T_i and was thus finished. Therefore there are $|T_{i+1}|$ jobs, each with remaining work of at least 2^i units each. How quickly might these all complete?

Since there are more than $|T_{i+1}|$ machines in the sets $G_{i+1}, G_{i+2}, \ldots G_k$, in an optimal schedule there is no need to run one of the jobs in T_{i+1} on a machine in G_i or slower. At best, processing all the jobs from T_{i+1} on all the machines in G_{i+1} and faster must take time at least the sum of the remaining processing requirements of the jobs in T_{i+1} over the sum of the processing speeds of processors in G_{i+1} or faster.

So the time taken is at least

$$\begin{aligned}
\frac{\sum_{j \in T_{i+1}} p_j/2}{\sum_{l \in \cup_{i+1}^k G_i} s_l} &= \frac{2^i |T_{i+1}|}{\sum_{l=i+1}^k s_l |G_l|} \\
&= \frac{2^{2k-i-3}}{\sum_{l=i+1}^{k-1} 2^l \cdot 2^{2k-2l-1} + 2^k} \\
&= \frac{2^{2k-i-3}}{2^k \sum_{r=0}^{k-i-2} 2^r + 2^k} \\
&= \frac{2^{2k-i-3}}{2^k(2^{k-i-1} - 1) + 2^k}
\end{aligned}$$

$$= \frac{2^{2k-i-3}}{2^{2k-i-1}}$$
$$= \frac{1}{4}.$$

■

The $\Omega(\log m)$ lower bound follows directly from these lemmas.

Theorem 3.4.12 The competitive ratio for any deterministic on-line algorithm for scheduling uniformly related machines with $R = s_1/s_m < m$ is $\Omega(\log R)$, whether or not preemption is allowed.

Proof: We modify our previous proof slightly. Let R' be the largest power of 2 less than or equal to R. We again let $k = (\log(3m-1)+1)/2$, and restrict ourselves to values of m such that k is integral. The instance $\mathcal{I}_k$ has k sets of machines G_i and k sets of jobs T_i, $1 \leq i \leq k$. $|G_i| = |T_i| = 2^{2k-2i-1}$ for $1 \leq i < k$, and $|G_k| = |T_k| = 1$. Once again each job in T_i has size 2^i; the only difference is that each machine in G_i has a speed of 2^i for $i \geq k - \log R'$; the machines G_i, $i < k - \log R'$, all have the same speeds as the machines in $G_{k-\log R'}$, namely $2^k/R'$. Thus s_1/s_m for this instance is R'. Again, the optimal schedule length is 1. By using the same strategy as in the previous proof, the adversary can force the scheduler to construct a schedule for $\mathcal{I}_k$ in which $X_1 \leq X_2 \leq \ldots \leq X_k$, and in which $X_i - X_{i-1} \geq \frac{1}{4}$, $k - \log R' \leq i \leq k$, $X_0 = 0$. All of the "small" jobs may finish quite quickly, but those in $T_{k-\log R'}, \ldots, T_k$ will each take at least $\frac{1}{4}$ unit of time to complete. ■

3.5 On-line Algorithms for Other Scheduling Models

3.5.1 Shop Scheduling

The juxtaposition of the results in this and the last chapter raises the question of how well can one schedule shop problems on-line; specifically, how well can one schedule if the processing time of each operation of a job is unknown until the completion of that operation, but the number of operations of each job and the order constraints on their execution are known in advance?

The most naive on-line strategy for shop problems is to arbitrarily choose jobs to process, subject to the condition that no machine sits idle when there is some operation of a job which it could be processing. We have already noted that for the flow shop and job shop problems such a strategy yields a m-approximation algorithm, and furthermore that this bound is tight. For the open shop problem we have proved that such a strategy is a 2-approximation algorithm, and have shown that it can do as badly as $(2 - \frac{1}{m})$. Our belief is that these bounds are close to the the best one can do on-line; we give one small theorem in that direction.

Theorem 3.5.1 No deterministic on-line algorithm for open shop scheduling has a competitive ratio better than $\frac{3}{2}$.

Proof: First consider a simple example with two machines and three jobs, and let A be a deterministic on-line open shop scheduling algorithm. Each job has an operation on both machines; the adversary will determine the sizes of these operations based on the strategy of A.

Assume A begins by starting two jobs on machines 1 and 2, without loss of generality job 1 on machine 1 and job 2 on machine 2. Let t be the first point in time at which A interrupts a job; if A never does let $t = 1$. We define job 1 to have an operation of size t on machine 1 and an operation of size 0 on machine 2; similarly job 2 is defined to have an operation of size t on machine 2 and an operation of size 0 on machine 1. Job 3 is defined to have an operation of size t on each machine. The optimal schedule length is $2t$, but A will not start job 3 until time t and will produce a schedule of length $3t$.

If A does not start two jobs at time 0 let t be the point at which it starts a job on the second machine. Let job 1 have an operation of size 1 on machine 1; let job 2 have an operation of size t on machine 2; let all other operations be of size 0. Then A produces a schedule of length at least $2t$ when the optimal length is t.

To extend this example to any even number m of machines, just duplicate the 2-machine example $\frac{m}{2}$ times. ■

We can also give the same lower bound in the on-line scheduling scenario where jobs have unknown release dates but once they arrive they are fully specified.

Theorem 3.5.2 No deterministic algorithm for open shop scheduling with unknown release dates but fully specified jobs (on arrival) has competitive ratio better than $\frac{3}{2}$.

Proof: We again give a simple two machine example that can be duplicated $\frac{m}{2}$ times. At time 0 there is just one job released, that has an operation of size 1 on each of the two machines. At time 1 release a job which has one operation of size 1 on whichever machine the scheduling algorithm left idle from time 0 to 1. The optimal length of a schedule for this instance is 2, but the algorithm will produce a schedule of length at least 3. ■

3.5.2 Precedence Constraints

As we discussed in section 1.3, an important element in a model of parallel processor scheduling is *precedence constraints* between the jobs, that form a partial order (dag) reflecting the logical flow of information in the program. We refer to the *levels* of the dag, where level 0 contains jobs that are not preceded by any other jobs. A reasonable *on-line* model of this element is that when a job arrives, despite the fact that its size is unknown, the precedence constraints between it and any other previously known-about job are known as well. Further, we must assume that when a job in level i of the dag arrives, the jobs in levels $0, \ldots, i-1$ of the dag have already arrived.

Graham proved in 1966 that list scheduling is a $(2-\frac{1}{m})$-approximation algorithm for scheduling identical machines even with precedence constraints. Since we have proved a lower bound of $(2-\frac{1}{m})$ without precedence constraints, $(2-\frac{1}{m})$ is a tight bound. For uniformly related machines, the current situation is much bleaker, since the best *off-line* approximation algorithm for uniformly related machines with precedence constraints is only an $O(\sqrt{m})$-approximation algorithm; this algorithm, however, happens to be an on-line algorithm as well.

3.6 Conclusions and Open Problems

The most obvious open problem raised by our work is to close the gap between the upper bound of $O(\log n)$ and the lower bound of $\Omega(\log m)$ for unrelated machines. All that would be necessary to do this would be a "preprocessing algorithm" that reduced the number of jobs to a number polynomial in m. For uniformly related machines, we showed that list scheduling of the first $n-m$ jobs accomplishes this goal. It is not clear, however, that a similar naive approach will be of use for unrelated machines.

Other issues we have not fully resolved are the usefulness of randomization in uniformly related and unrelated machines, and an exact characterization of the power of the on-line shop scheduler. Another direction would be average case analyses of on-line algorithms for these problems. With regard to scheduling of parallel identical machines, Bruno and Downey [18] proved that when the p_j are independently and uniformly distributed over the interval $[0,1]$,

$$\lim_{n\to\infty} \text{Prob}[\max_j p_j / \sum_j p_j > \frac{4}{n}] = 0.$$

This implies that when n grows faster than m, list schedules are asymptotically optimal with probability that goes to 1. Coffman and Gilbert refined and extended these results to other distributions [35]. However, in contrast to the relative error, the absolute error does not tend to 0 as $n \to \infty$, with m fixed. This stronger criterion of optimality is satisfied by the (off-line) algorithm that schedules jobs in order of longest processing time. It would be interesting to prove that no on-line algorithm can do better than list scheduling in this regard; it would also be interesting to carry out similar analyses of on-line algorithms for uniformly related and unrelated machines.

We mentioned earlier, in the context of paging and list maintenance, that the conclusions drawn from average-case analysis of on-line algorithms do not always correspond to the conclusions of experimental studies and practical experience. If this proves to be the case for parallel machine scheduling, then the design and analysis of a model that is less pessimistic than the worst-case competitive ratio but has more structure than expected performance on randomly-selected instances might be a valuable endeavor.

Chapter 4

Parallel Network Optimization: An Introduction

In the last decade there has been much interest in harnessing the power of parallel computers to solve large complex problems in real time. The first step in any such effort must be to understand how efficiently the most basic problems can be solved by parallel computers, and to then construct more complex systems out of these building blocks. Accordingly both theoreticians and practitioners have put much effort into the study of parallel algorithms for a large variety of basic problems.

In the field of combinatorial optimization one can hardly find more basic problems than the matching, maximum flow and minimum-cost flow problems. Many of the important concepts that arose out of the study of these problems – augmenting paths, scaling, relaxed optimality, strong polynomiality – have been used widely in other areas of combinatorial optimization. Each has a large number of important applications in and of itself; furthermore these problems serve as building blocks for a wide variety of more complex applications, such as the traveling salesman problem [80], cyclic staff scheduling, machine scheduling [83] and vehicle and crew routing [16]. We therefore expect that the study of parallel algorithms for these problems will yield important insights into the theory of parallel computation while also providing useful building blocks for the parallel solution of harder problems.

In the next three chapters of this thesis we consider several practical and theoretical issues about parallel algorithms for these problems. In this chapter we introduce this topic by formally defining the problems and our theoretical model of parallel computation, and by discussing previous theoretical work on parallel algorithms for these problems.

4.1 Definitions and Models

In the matching problem we are given an undirected graph $G = (V, E)$, possibly with an associated weight function $w : E \rightarrow \mathcal{Z}^+ \cup \{0\}$. Let $w_{\max} = \max_{e \in E} w(e), n = |V|$ and $m = |E|$. Given $S \subseteq E$, define $\deg_S(v)$ to be the degree of v in the graph (V, S). A *matching* in a graph G is a set of edges $M \subseteq E$ such that no two edges in M share a common endpoint; i.e., $\deg_M(v) \leq 1$, $\forall v \in V$. A *perfect matching* is a matching of cardinality $\frac{|V|}{2}$. A *minimum weight perfect matching* is a perfect matching M that minimizes $\sum_{e \in M} w(e)$. If G is bipartite then a perfect matching is known as an *assignment*. In various forms of the problem one is asked to produce, for example, a maximum cardinality matching, a maximum-weight matching or a minimum-weight perfect matching.

In the *maximum-flow* problem we are given a *flow network* $G = (V, E)$, which is a directed graph with two distinguished vertices, s and t, where s is called the source and t the sink. With every edge $e = (i, j)$ of a flow network is associated a capacity $u(i, j) \geq 0$. A *flow* is a real valued function $f : E \rightarrow \mathbf{R}^+ \cup \{0\}$ that satisfies the following two constraints:

Capacity Constraint: For all $i, j \in V$, we require $f(i, j) \leq u(i, j)$.

Flow conservation: For all $v \in V$, $v \notin \{s, t\}$,

$$\sum_{\substack{i \in V \\ (i,v) \in E}} f(i, v) = \sum_{\substack{j \in V \\ (v,j) \in E}} f(v, j)$$

The *value of a flow* is defined as

$$\sum_{\substack{i \in V \\ (i,t) \in E}} f(i, t);$$

a *maximum flow* is simply a flow of maximum value.

The *minimum-cost flow* problem is the weighted generalization of the maximum-flow problem. We assign a cost function $c : E \rightarrow \mathbf{R}$ to the edges of G; the *cost* of a flow f is defined as the sum

$$\sum_{(i,j)\in E} f(i,j)c(i,j).$$

We can then ask for the "minimum-cost flow" or the "minimum-cost maximum flow."

Models of Parallel Computation

Our theoretical model of parallel computation will be the CRCW PRAM [73]. A PRAM consists of a number of sequential independent processors, each with its own private memory, communicating with one another through a global memory. In one unit of time, each processor can read one global or local memory location, execute a single RAM operation, and write into one global or local memory location. In the CRCW PRAM we allow concurrent reads of the same memory location, and concurrent writes to the same location; conflicts in writes are resolved by a a priority mechanism: the write of the processor with the lowest number succeeds. We mention the details of the PRAM here for the sake of completeness; when we describe our parallel algorithms in chapter 6 we will do so at a fairly high level of detail.

The complexity classes which correspond to our notion of *easy to parallelize* are $\mathcal{NC}$ and $\mathcal{RNC}$. $\mathcal{NC}$ is the class of decision problems for which there exist algorithms that run in time $O(\log^k n)$ on a CRCW PRAM with $O(n^c)$ processors, where c and k are constants and n is the size of the input. $\mathcal{RNC}$ is the corresponding class of decision problems with *randomized* algorithms that run in time $O(\log^k n)$ on a CRCW PRAM with $O(n^c)$ processors and produce the correct answer with probability at least $\frac{2}{3}$.

By a *decision problem* we refer to a set of instances of a problem, all of whom satisfy some property. For example, the perfect matching decision problem would be the set of all graphs G that contain a perfect matching; the decision question would be "Does G contain a perfect matching?" Another example is the maximum-flow problem; the decision problem might be the set of all flow networks N for whom the value of the maximum flow is odd; the decision question would be "Is the value of the maximum flow in the flow network N odd?" The *optimization* versions of these problems would not ask for a True/False answer, but rather a value, such as the size of the maximum matching or the value of the maximum flow. We will at times blur the distinction between the two. For example, in chapter 6 we give randomized algorithms to actually find minimum (unary) weight matchings. Since these algorithms run in time $O(\log^k n)$

on a CRCW PRAM with $O(n^c)$ processors, where c and k are constants and n is the size of the input, we will say the problem is in $\mathcal{RNC}$; it is clear how to use the optimization algorithm as a decision algorithm. For a fuller discussion of the classes $\mathcal{NC}$ or $\mathcal{RNC}$, see the survey of Karp and Ramachandran [73].

The complexity class that corresponds to our notion of *difficult to parallelize* is the class of $\mathcal{P}$-Complete decision problems [73]. Analogous to the $\mathcal{NP}$-Complete problems in their role as the "hardest" problems in the class $\mathcal{P}$, no $\mathcal{NC}$ or $\mathcal{RNC}$ algorithm for any $\mathcal{P}$-Complete problem exists unless $\mathcal{P}$ is equal to, respectively, $\mathcal{NC}$ or $\mathcal{RNC}$.

4.2 Previous Work

The parallel computational complexity of the matching problem is one of the most interesting open questions in the theory of parallel computation today. The first major progress was made by Karp, Upfal and Wigderson, who gave an $\mathcal{RNC}$ algorithm to find a maximum matching with an $O(\log^3 n)$ worst-case time bound in the PRAM model [70]. Mulmuley, Vazirani and Vazirani improved this result to an $\mathcal{RNC}$ algorithm with an $O(\log^2 n)$ worst case time bound. These algorithms were *Monte Carlo* randomized algorithms, meaning that with high probability they produced a correct answer, but could not indicate when they produced an incorrect answer. A desirable goal is a *Las Vegas* algorithm: an algorithm that with high probability produces a correct solution and otherwise outputs the word FAILURE. In this case one can try again and eventually arrive at the optimal solution.

Karloff [69] gave a Las Vegas $\mathcal{RNC}$ algorithm for the maximum matching problem by utilizing a "min-max" duality relation, the Tutte-Berge formula [104], that characterizes the size of a maximum matching. Karloff showed that the minimum side of the relation could be calculated in $\mathcal{RNC}$; if this is equal to the size of the candidate maximum matching the Tutte-Berge formula guarantees that they are both optimal.

The results of Karp, Upfal and Wigderson and of Mulmuley, Vazirani and Vazirani both yielded algorithms for the weighted versions of the problem as well. Since the number of processors is proportional to $n^{3.5}mw_{\max}$, where $w_{\max}$ is the maximum edge weight, these are only $\mathcal{RNC}$ algorithms if the edge weights are input in unary. No $\mathcal{RNC}$ algorithm is known for the maximum weight matching problem or the minimum weight perfect matching problem when the weights are input in binary; nor are these problems known to be $\mathcal{P}$-Complete. Further, no Monte Carlo $\mathcal{RNC}$ algorithms for unary weighted matching problems were known until our work.

Although no deterministic $\mathcal{NC}$ algorithms are known for matching problems, some progress has been made on deterministic sublinear time parallel algorithms. We say that an algorithm runs in $O^*(f(n))$ time if it runs in $O(f(n)\log^k n)$ time for some constant k. Goldberg, Plotkin and Vaidya [46] gave a parallel algorithm to find a maximum matching in a bipartite graph that ran in $O(n^{\frac{2}{3}})$ time using $BFS(n,m)$ processors, and a parallel algorithm to find a minimum-weight assignment that ran in $O^*(n^{\frac{2}{3}}\log(nC))$ time using $SSP(n,m)$ processors. Here $C = w_{\max}$ is the maximum edge cost, $BFS(n,m)$ denotes the maximum of $n+m$ and the number of processors required to find a breadth-first search tree in $O(\log^2 n)$ time, and $SSP(n,m)$ denotes the maximum of $n+m$ and the number of processors required to find a single-source shortest-path tree (with non-negative weights) in $O(\log^2 n)$ time.

Goldberg, Plotkin, Shmoys and Tardos [48] used interior point methods to give an $O^*(\sqrt{m}\log C)$ time parallel algorithm for the assignment problem and an $O^*(\sqrt{m})$ time algorithm for the unweighted version. Both of these algorithms used $O(m^3)$ processors.

Besides the running time, another way to measure the performance of a parallel algorithm is the total *work* performed, where we define work as the product of processors $\times$ time. The best results for bipartite matching and the assignment problem in this regard are those of Gabow and Tarjan, who for a large range of numbers of processors p give parallel algorithms for p processors with total work within a factor of $O(\log p)$ of the work of the best sequential algorithm [43].

As a consequence of the $\mathcal{RNC}$ matching algorithms, the maximum-flow problem with the edge capacities input in unary is also in $\mathcal{RNC}$. The maximum-flow problem and the minimum-cost flow problem are known to be $\mathcal{P}$-Complete, however, when the edge capacities are input in binary [49]. Therefore it is believed that there exist no $\mathcal{NC}$ or $\mathcal{RNC}$ algorithms for these problems. Nonetheless parallel computation can yield speedups in the running time; for example the Goldberg-Tarjan maximum-flow algorithm can be implemented so as to give an $O(n^2\log n)$ time, $O(n)$ processor algorithm [45]. For the minimum-cost flow problem they give a parallel algorithm that takes $O(n^2(\log n)\log(nC))$ time on $O(n)$ processors [45].

In the next three chapters we present three results about parallel algorithms for these problems. In Chapter 5 we give a simple but interesting separation between the parallel complexity of the maximum-flow problem and the minimum-cost maximum flow problem, showing that the minimum-cost maximum flow problem can not be approximated in $\mathcal{NC}$ or $\mathcal{RNC}$ unless $\mathcal{P}$ is equal to, respectively, $\mathcal{NC}$ or $\mathcal{RNC}$. In contrast, it is known that the maximum-flow problem can be approximated quite closely in $\mathcal{RNC}$, and that by a very recent result of Fischer, Goldberg and Plotkin, that the maximum matching problem can be approximated arbitrarily closely in $\mathcal{NC}$ [42].

In Chapters 6 and 7 we consider both theoretical and experimental issues in the parallel solution of weighted matching problems. In chapter 6 we give *Las Vegas* $\mathcal{RNC}$ algorithms to find a minimum-weight perfect matching when the edge weights are input in unary. We also show how to apply the technique to a number of other problems as weli.

In Chapter 7 we describe an experimental study of various parallel algorithms for the assignment problem on a real massively parallel computer, the Connection Machine CM-2[1]. We consider in detail one special case, that of the fully dense assignment problem, where the graph is a complete graph. We implemented a number of different algorithms for this problem, and developed a new hybrid approach similar to the algorithm of Goldberg, Plotkin and Vaidya. This proved to be the most successful by a considerable margin. We also discuss the viability of the other major approaches which we did not implement.

[1]This is a trademark of Thinking Machines Corporation.

Chapter 5

The Parallel Approximability of a Flow Problem[1]

Once a problem is proved to be $\mathcal{P}$-complete, it is generally believed that there exists no $\mathcal{NC}$ or $\mathcal{RNC}$ algorithm to solve it exactly[2]. Therefore, the next important question becomes how well can it be *approximated* in $\mathcal{NC}$ or $\mathcal{RNC}$? In this chapter we show that despite the fact that one can approximate the value of a maximum flow arbitrarily closely in $\mathcal{RNC}$, approximating the value of the minimum-cost maximum flow, within a factor of the maximum cost in the network, is $\mathcal{P}$-Complete. Our proof also shows that this is true for networks with maximum cost polynomial in the size of the network. The chapter consists of two short sections. In Section 5.1 we discuss previous work on the $\mathcal{NC}$-approximability of $\mathcal{P}$-complete problems and flow problems in particular. In Section 5.2 we prove our result.

5.1 Background

There has been some amount of work on $\mathcal{NC}$-approximation algorithms for $\mathcal{P}$-Complete problems. For example, Anderson and Mayr considered the High Degree Subgraph Problem: Given a graph G and an integer k, find the maximum induced subgraph of G that has all nodes of de-

[1]This chapter describes joint work with Cliff Stein [117].

[2]Despite the fact that $\mathcal{P}$-Completeness is usually defined in terms of decision problems, in this chapter we will often refer to the optimization versions as well. This has no effect on our results.

gree at least k. They prove that this problem is $\mathcal{P}$-Complete, and further that it is $\mathcal{P}$-Complete to approximate within any factor better than $\frac{1}{2}$ [2]. In other words, it is $\mathcal{P}$-Complete to produce a subgraph that is of size greater than $\frac{1}{2}$ of the maximum induced subgraph with the appropriate connectivity constraints. In contrast to this result they give an $\mathcal{NC}$ algorithm that can approximate the solution within a factor arbitrarily close to $\frac{1}{2}$. Subsequently Hochbaum and Shmoys showed how to approximate it within a factor of exactly $\frac{1}{2}$ [59]. A similar result was obtained by Kirousis, Serna and Spirakis [76], who investigated the High Edge-Connectivity Subgraph Problem, and showed it could be approximated in $\mathcal{NC}$ within any factor $< \frac{1}{2}$, but producing a better approximation was $\mathcal{P}$-Complete. They also demonstrated the same type of behavior for the vertex-connectivity version of the problem.

There have also been several results that show that a certain $\mathcal{P}$-complete problem can not be approximated in $\mathcal{NC}$ within any factor unless $\mathcal{P} = \mathcal{NC}$ [77, 107]. The most interesting of these is a recent result by Serna, who proved the $\mathcal{P}$-Completeness of approximating Linear Programming within any factor [106]. This raises the question of whether the same is true for other $\mathcal{P}$-Complete problems, such as maximum flow or minimum-cost flow, that can be described as combinatorial linear programs.

It is unlikely that such a result is true for the maximum-flow problem, since it is known that it can be approximated arbitrarily closely in $\mathcal{RNC}$. This is a consequence of the unary capacity version of the problem being in $\mathcal{RNC}$. A binary capacity problem instance Ψ can be approximated within a factor of $(1 + \frac{1}{\text{polynomial}(n)})$ by using the unary capacity algorithm on a scaled version of Ψ whose capacities are only the high order $O(\log n)$ bits of the binary capacities. It is also true that the minimum-cost maximum flow problem is in $\mathcal{RNC}$ when both the capacities and the costs are in unary. Therefore one might imagine that it might be possible to approximate the minimum-cost maximum flow problem with binary capacities and unary costs in $\mathcal{RNC}$; we prove that this is not the case unless $\mathcal{P} = \mathcal{RNC}$.

5.2 Proof

Definition 5.2.1 *$AMCF(\rho)$* is the problem of approximating the value of the minimum-cost maximum flow in a network to within a factor of ρ, where the capacities are expressed in binary and the costs are expressed in unary.

Theorem 5.2.2 *AMCF(ρ)* is log-space complete for $\mathcal{P}$.

We will prove this theorem by reducing a form of the monotone circuit value problem (*MCV2*) to *AMCF(ρ)*. The reduction is a simple generalization of the proof of Goldschlager, Shaw and Staples [49] that the problem of determining the *exact* value of the maximum flow in a network is log-space complete for $\mathcal{P}$. Before we begin the proof we give several necessary definitions and known results.

Definition 5.2.3 A monotone circuit α is a sequence $(\alpha_n, \ldots, \alpha_0)$ where each α_i is either an input, in which case its value of either 0 or 1 is given explicitly, or a gate. A gate α_i is either an AND gate AND(j,k) or an OR gate OR(j,k) where $j \geq k > i$. The fan-out of a gate α_j is the number of gates α_k, $k < j$, to which α_j is an input.

Definition 5.2.4 *MCV2* is the problem of determining the value of a monotone circuit such that each input has fan-out at most one, each gate has fan-out at most 2, and the last gate is an OR gate.

Theorem 5.2.5 [49] *MCV2* is log-space complete for $\mathcal{P}$.

We are now ready to begin the proof of our theorem.

Proof of Theorem 5.2.2: Let $A = (\alpha_n, \ldots, \alpha_1)$ be an instance of $MCV2$, and let d_i be the fan-out of gate (or input) α_i. We will demonstrate a log-space transformation of A into a flow network $G_A = (V, E)$. The vertices of G_A are s, t, and i, $0 \leq i \leq n$. The edges of G_A, and their capacities and costs, are as follows.

Type 1 Cost 0 edges:

- For each input α_i include an edge (s,i) of capacity 0 if α_i is false, or of capacity 2^i if α_i is true. Also include an edge (i,s) of capacity 2^i.
- For every OR gate $\alpha_i = \mathrm{OR}(j,k)$ include an edge (j,i) of capacity 2^j, (k,i) of capacity 2^k, and (i,s) of capacity $2^j + 2^k - d_i 2^i$.

- For every AND gate $\alpha_i = \text{AND}(\alpha_j, \alpha_k)$, include an edge (j,i) with capacity 2^i, an edge (k,i) with capacity 2^k, and an edge (i,t) with capacity $2^j + 2^k - d_i 2^i$.

Type 2 An edge $(0,t)$ with capacity 1 and cost ρ.

Type 3 An edge (s,t) with capacity 1 and cost 1.

Goldschlager, Shaw and Staples showed that in the flow network composed of the edges of type 1 and 2, the maximum flow value is odd if and only if the circuit A outputs true. Furthermore the maximum flow value is odd if and only the edge $(0,t)$ has one unit of flow. Therefore to determine the output of circuit A we merely have to determine the flow on edge $(0,t)$ in a maximum flow. We will show how to prove this momentarily. Assuming for the moment that it is true, we first complete the proof of our theorem.

Notice that given our cost assignment, the cost of a maximum flow is $\rho + 1$ if there is one unit of flow on edge $(0,t)$ and is otherwise 1. The edge (s,t) will be have one unit of flow in any maximum flow and this edge will therefore contribute a cost of 1. Edge $(0,t)$ will contribute ρ units to the cost if edge $(0,t)$ has one unit of flow on it.

Therefore if we could approximate the minimum-cost maximum flow problem within a factor of ρ we could determine whether the value for this network was 1 or ρ thus determine the parity of the maximum flow in the underlying network which gives the output of circuit A. This reduction is certainly in logspace as long as ρ is polynomial in the size of the circuit n. If we wish to allow edge costs to be expressed in binary, ρ can be exponential in the size of the network.

We will now explain the proof that circuit A outputs true if and only if the value of the maximum flow in the flow network that contains the edges of type (1) and (2) from G_A. Call that network $G_{A'}$. We will exhibit a flow f in $G_{A'}$ and then prove that it is a maximum flow. We denote the flow on edge (x,y) by $f(x,y)$ and the capacity of edge (x,y) by $u(x,y)$.

In flow f, for $0 \leq i \leq n$,

- For α_i an input of circuit A, $f(s,i) = u(s,i)$. If α_i is not an input to any other gate, $f(i,s) = u(i,s)$; otherwise it is zero.
- For $0 \leq j \leq n$, $f(i,j) = 2^i$ if gate α_i outputs true, otherwise $f(i,j) = 0$.

- If $\alpha_i = \text{AND}(j,k)$ then $f(i,t) = u(i,t)$ if both α_j and α_k output true. Otherwise $f(i,t) = f(j,i) + f(k,i)$. The intuition here is that the node representing α_i, in a maximum flow, will only have $d_i 2^i$ units of flow to send from i if it receives 2^j and 2^k as inputs. Since this is a maximum flow, all flow will be directed onto (i,t) until that arc is saturated. Only if both gates α_k and α_j output true will there be flow "left-over" after (i,t) has been saturated.

- If $\alpha_i = \text{OR}(j,k)$ then $f(i,s) = f(j,i) + f(k,i) - d_i 2^i$ if either α_j or α_k outputs true. The intuition here is that in a maximum flow flow will go anywhere before going back to s, so if any flow is input from j or k it will become the output of i before returning to s.

- $f(0,t) =$ if α_0 computes true; otherwise $f(0,t) = 0$.

The parity of the value of f is odd if and only if the circuit A outputs true. This is because f assigns an even amount of flow to every edge except perhaps $(0,t)$, which is 1 if and only if A outputs true. It remains to prove that f is a maximum flow. This can be proved in the standard way, by showing there is no augmenting path. If there is an augmenting path from s to t then the first edge must be a back edge, since each forward edge (s,i) out of s has $f(s,i) = u(s,i)$. It must end with a forward edge since there are no edges out of t. Therefore somewhere on the path there must be a back edge followed by a forward edge. A simple case analysis shows however that this is impossible. ■

Chapter 6

Las Vegas $\mathcal{RNC}$ Algorithms

6.1 Introduction

In this chapter we present a Las Vegas $\mathcal{RNC}$ algorithm for the problem of finding a minimum-weight perfect matching in a graph when the weights of the edges are input in unary. We utilize a transformation of minimum-weight perfect matching to the T-join problem, and use the structure theory of T-joins as developed by Sebö [105] to develop an optimality condition that can be computed in $\mathcal{RNC}$. Easy consequences of this result are Las Vegas $\mathcal{RNC}$ algorithms for finding a maximum weight matching, a half-integral planar multicommodity flow, a minimum weight T-join. a maximum 1-packing of T-cuts in a bipartite graph and the T-join structure of a graph.

6.2 Background

6.2.1 T-joins

We are given an undirected graph $G = (V, E)$ with an associated weight function $w : E \to \mathcal{Z}^+ \cup \{0\}$. Let $T \subseteq V$ with $|T|$ even. A set $F \subseteq E$ is called a *T-join* if $\deg_F(x) = 0(\text{mod } 2)$ for $x \notin T$, and $\deg_F(x) = 1(\text{mod } 2)$ for $x \in T$. A *minimum T-join* is a T-join of minimum cardinality. We define $\tau(G,T)$, $|T|$ even, to be the size of the minimum T-join for G. A *minimum weight T-join F* is a T-join which minimizes $\sum_{e\in F} w(e)$.

T-joins can be viewed as generalizations of matchings, Chinese postman tours, and paths. For example, an s to t path is a T-join with $T = \{s,t\}$ and a perfect matching M is a V-join. Or, consider the Chinese postman problem: find a minimum length tour that traverses every edge of a graph at least once. In an Eulerian graph this is just an Eulerian tour; in a general graph there is a clear one-to-one correspondence between minimum postman tours and minimum T-joins, where T is the set of all odd degree nodes in G [105].

The following well-known facts will be important; therefore, we present their proofs here.

Fact 6.2.1 [86, 105] The problem of finding a minimum weight perfect matching in an arbitrary graph whose edge weights are input in unary can be reduced in $\mathcal{NC}$ to finding a minimum cardinality T-join in a bipartite graph.

Fact 6.2.2 [86] The problem of finding a minimum cardinality T-join can be reduced in $\mathcal{NC}$ to finding a minimum weight perfect matching in a graph whose edge weights are represented in unary.

Proof of Fact 6.2.1

We first reduce finding a minimum weight perfect matching to finding a minimum weight T-join in an arbitrary graph. Let $T = V$, $\tilde{w}(e) = w(e) + nw_{\max}$, and let $\tilde{G}$ be G with edge weights $\tilde{w}$. Note that $\tilde{w}(e) > 0\ \forall e$. If $\tilde{G}$ has a perfect matching, it has a T-join of weight not exceeding $\frac{n}{2}(w_{\max} + nw_{\max})$. Any other T-join has weight exceeding $(\frac{n}{2} + 1)nw_{\max}$, so if $\tilde{G}$ has a perfect matching, the minimum weight T-join in $\tilde{G}$ is a minimum weight perfect matching in G.

Finding a minimum weight T-join in a graph G can be reduced to finding a minimum cardinality T-join in a graph G^*, where we replace an edge $e = pq$ of weight $w(e)$ by a path $pv_1v_2\cdots v_{w(e)-1}q$. None of the v_i are included in T. It is easy to see that there is a one-to-one correspondence between minimum cardinality T-joins in G^* and minimum weight T-joins in G.

Given a non-bipartite graph G, we can construct a bipartite graph G' whose T-joins easily correspond to those of G, by inserting a vertex v_e in the middle of each edge e and not including v_e in T.■

Proof of Fact 6.2.2:

Compute the shortest paths between each pair of vertices in T; it is well known this can be done in $\mathcal{NC}$ [73]. Now consider the complete graph with vertex set T where the weight of edge ij is the length of the shortest path from i to j in G. If we find a minimum weight perfect matching M in this graph, the set of edges of G that correspond to the paths represented by the edges of M form a minimum cardinality T-join. Note that if G has n vertices the maximum edge weight arising from this shortest-paths computation is n and therefore can be represented in unary while increasing the size of the problem by a most a polynomial in its original size; therefore the reduction can be carried out in $\mathcal{NC}$. ■

6.2.2 T-cuts

In this section we describe a dual to the T-join, and min-max relations that will allow us to certify when both are optimal.

For $X \subseteq V$ let $\delta(X) = \{xy \in E : x \in X, y \notin X\}$. The subset $K \subseteq E$ is a *cut* if $K = \delta(X)$ for some $X \subset V$. If $|T \cap X| \equiv 1 (\text{mod } 2)$, then $\delta(X)$ is called a *T-cut*.

A *k-packing of T-cuts*, where k is a positive integer, is a multiset Ψ of T-cuts such that each edge in the graph appears in no more than k of the cuts. We define
$\nu_k = \nu_k(G,T) = \max\{|\Psi| : \Psi \text{ is a } k\text{-packing of } T\text{-cuts}\}$.

It is not hard to see that $\tau \geq \frac{\nu_2}{2} \geq \nu_1$. The following minimax theorems are known.

Theorem 6.2.3 [85] Let G be a graph; then $\tau(G,T) = \frac{\nu_2(G,T)}{2}$.

Theorem 6.2.4 [111] Let G be a bipartite graph; then $\tau(G,T) = \nu_1(G,T)$.

6.2.3 Sketch of Algorithm

Theorem 6.2.4 implies that if we are given a T-join F and a 1-packing of T-cuts Q in a bipartite graph such that $|F| = |Q|$, then F is a minimum T-join and Q is a maximum 1-packing of T-cuts. It is this property we use in our algorithm, which we now sketch.

1. Given G, use the reduction of Fact 6.2.1 to reduce the problem of finding a minimum weight perfect matching in G to that of finding a minimum cardinality T-join in bipartite $\tilde{G}$. Then use Fact 6.2.2 and the randomized matching algorithm of Mulumuley, Vazirani and Vazirani to find a minimum unweighted T-join F in $\tilde{G}$. With high probability F is a minimum cardinality T-join in $\tilde{G}$, but with low probability it is not. (We actually have to

test that it is a T-join at all, since if the algorithm fails the result need not be a T-join. We ignore this detail for the rest of the chapter.) Call F the *candidate* minimum T-join.

2. Calculate $\nu_1(\tilde{G}, T)$ in $\mathcal{RNC}$. How this is done will be discussed below, but the important point is that if ν_1 is the returned value with high probability it is indeed $\nu_1(\tilde{G}, T)$ and with low probability it is not. Call the packing that is found the *candidate maximum packing.*

3. If $|F|$ is the size of our candidate T-join, and ν_1 is the size of our candidate 1-packing of T-cuts and $|F| = \nu_1$, then each was calculated correctly and is optimal. Reverse the reductions to produce a certified minimum-weight perfect matching.

We need to explain how to do Step 2 in $\mathcal{RNC}$. In order to calculate $\nu_1(\tilde{G}, T)$, we need to understand the structure theory of Sebö.

6.3 A Structure Theorem of Sebö

In this section we will present a theorem of Sebö that will enable us to construct a maximum 1-packing of T-cuts in a bipartite graph $G = (V, E)$ by calculating $|V|$ different T'-joins. Given a graph G and a set $T \subseteq V$, a necessary condition for G to possess a T-join is that $|T|$ be even. For all $x \in V$, define the set $S^x \subseteq V$ as follows

$$S^x = \begin{cases} S \cup \{x\} & \text{if } x \notin S \\ S - \{x\} & \text{if } x \in S. \end{cases}$$

Note that when $S \subseteq V$, $|S|$ odd, the S^x are all of even cardinality. A set of minimum cardinality S^x-joins for all $x \in V$ is called the *S − join structure* of the graph. It is such a set of S^x-joins we will use to construct a maximum 1-packing of T-cuts.

Theorem 6.3.1 [105] Let $G = (V, E)$ be a bipartite graph, $S \subseteq V$, $|S|$ odd. For each node $x \in V$ let F^x be a S^x-join. For all $x \in V$ define $\pi(x) = |F^x|$. Let G^i be the graph induced by the set of vertices x such that $\pi(x) \leq i$. Let $\mathcal{D}(\pi) = \{D : D \text{ is the vertex set of a connected component of } G^i \text{ for some } i\}$.

Then F^x is a minimum cardinality S^x-join for all $x \in V$ if and only if

1. $|\pi(y) - \pi(x)| = 1$ for all $xy \in E(G)$.
2. For all $x \in V$ and $D \in \mathcal{D}(\pi)$,

$$|F^x \cap \delta(D)| = \begin{cases} 0 & \text{if } x \in D \\ 1 & \text{if } x \notin D. \end{cases}$$

We will in a moment explain part of Sebö's proof of this theorem in order to explain how to construct the packing of T-cuts. We note first, however, that merely the statement of this theorem is enough information to yield our main result, a Las Vegas $\mathcal{RNC}$-algorithm for the minimum (unary) weight perfect matching problem[1]. We have already reduced this problem to finding a minimum-cardinality T-join in a bipartite graph. A Las Vegas $\mathcal{RNC}$ algorithm for that problem is as follows,

- If $T \neq \phi$ choose $v \in T$ arbitrarily and let $S = T^v$. If $T = \phi$ choose $v \in V$ arbitrarily and let $S = \{v\}$. In either case, note that $S^v = T$.
- Find $|V|$ (candidate) minimum S^v-joins, one for each $v \in V$, using Fact 6.2.2 and the $\mathcal{RNC}$ weighted matching algorithm.
- Apply Theorem 6.3.1 to verify that all the S^v are simultaneously optimal.

If the probability of failure in the calculation of each S^v-join is suitably small then this will be a Las Vegas $\mathcal{RNC}$-algorithm for a minimum cardinality T-join in a bipartite graph. This algorithm requires no explicit knowledge of T-cuts and is therefore conceptually simpler, but it is no more efficient than our main algorithm, and it also does not yield the the algorithmic results on packings of T-cuts in Corollary 6.4.2 nor the results of Section 6.5. Therefore we next give the proof that conditions 1 and 2 imply that each F^x is a minimum S^x-join in order to describe how a 1-packing of S^x-cuts is constructed.

Condition 1 implies that no edge connects two vertices that have the same π value. Therefore, every edge in E is in some cut $\delta(D), D \in \mathcal{D}(\pi)$. Furthermore, since $|\pi(y) - \pi(x)| = 1$, xy is in only one cut $\delta(D)$; therefore, $\{\delta(D) : D \in \mathcal{D}(\pi)\}$ is a partition of E. This enables

[1]We would like to thank the anonymous referee who made this observation.

us to express F^x as the union of its intersection with the members of $\mathcal{D}(\pi)$, and therefore $|F^x| = \sum_{D\in\mathcal{D}(\pi)} |F^x \cap \delta(D)| = |\{D \in \mathcal{D}(\pi) \ : x \notin D\}|$.

We claim that if $x \notin D$ then D contains an odd number of nodes in S^x, and that therefore $\delta(D)$ is a S^x-cut. This is true since a S^x-join is the union of cycles and simple paths between nodes in S^x. Therefore, if $|\delta(D) \cap F^x| = 1$, there must be exactly one of these simple paths that crosses the cut $\delta(D)$, and no cycles. Let $w \in T$ be the endpoint of that path in D. Since the degree of any node $v \in S^x$ must be odd in a S^x-join, there must be an even number of nodes in D besides w that are in S^x, since they must be paired up by simple paths that do not leave D. Thus there are an odd number of nodes of S^x in D, and $\delta(D)$ is a S^x-cut. Thus the set $\{\delta(D)|D \in \mathcal{D}, x \notin D\}$ is a 1-packing of S^x-cuts, which we showed earlier has cardinality $|F^x|$. By Theorem 6.2.4 each of the F^x is a minimum S^x-join and $\{\delta(D)|D \in \mathcal{D}, x \notin D\}$ is a maximum 1-packing of S^x-cuts. This completes the proof that conditions 1 and 2 imply that each F^x is a minimum S^x-join; furthermore, it shows how, from a complete set of F^x, $x \in V$, we can calculate $\nu_1(G, S^x)$ for any x.

6.4 An Algorithm for Certifying a Minimum Weight Perfect Matching

The one step of the algorithm from Section 6.3 still to be explained is the calculation, with high probability, of $\nu_1(\tilde{G}, T)$. Let $S = T - \{x\}$ for some arbitrary $x \in T$, and let S be the odd cardinality set referred to in Theorem 6.3.1. Let $\tilde{G}$ have $\tilde{n}$ nodes and $\tilde{m}$ edges. The parallel algorithm to calculate $\nu_1(\tilde{G}, T)$ is as follows:

1. Calculate, with high probability, $\pi(v)$ for all vertices v in $\tilde{G}$.

2. Calculate the connected components of $\tilde{G}^i$ for each i.

3. Calculate $|\{D \in \mathcal{D}(\pi) \ : x \notin D\}|$.

If the calculation in Step 1 succeeded, by the proof of Theorem 6.3.1 we know that $|D \in D(\pi) : x \notin D|$ is the size of the minimum cardinality S^x-join (T-join) in $\tilde{G}$. If this is equal to the size of the candidate minimum T-join, we know the candidate is optimal as is our original minimum weight perfect matching.

Each $\pi(v)$ can be calculated by finding a minimum cardinality S^v-join, which by Fact 6.2.2 can be done by a shortest paths computation and by finding a minimum weight prefect matching. Therefore Step 1 is in $\mathcal{RNC}$. Steps 2 and 3 are well known to be in $\mathcal{NC}$ [73], so the calculation of $\nu_1(\tilde{G}, T)$ can be carried out in $\mathcal{RNC}$.

Now consider the time and processor requirements in a CRCW PRAM model of computation [73]. In the reduction of finding a minimum weight perfect matching in G to finding a minimum cardinality T-join in $\tilde{G}$ we increase the size of the graph considerably. If G had n nodes, m edges and maximum edge weight $w_{\max}$, we increase the edge weights to $O(nw_{\max})$ and then expand each edge of weight w into an unweighted path of length w. Thus the number of nodes in $\tilde{G}$, $\tilde{n}$, is $O(nmw_{\max})$ and the number of edges, $\tilde{m}$ is $O(nmw_{\max})$ as well.

In Step 1 we must find $\tilde{n}$ S^v-joins in $\tilde{G}$. In order to find each of these $O(nmw_{\max})$ S^v-joins we compute a minimum weight perfect matching in a graph with n nodes, $O(n^2)$ edges, and maximum edge weights of $O(nm \cdot w_{\max})$. This is because $|T| = n$ both in G and in $\tilde{G}$, and the length of a path in $\tilde{G}$ is bounded by $\tilde{m}$. The algorithm of Mulmuley, Vazirani and Vazirani requires $O(N^{3.5}MW)$ processors and $O(\log^2 N)$ time to find a minimum weight perfect matching in a graph with N nodes, M edges and maximum edge weight W. A factor of NM in the processor bound comes from scaling up the edge weights by NM in order to ensure that the probability of success is at least $\frac{1}{2}$. In our reduction to the T-join problem we have already scaled up the edge weights by a factor of n. Since we are doing $O(nmw_{\max})$ calculations, however, we require a smaller probability of failure for each calculation, specifically at most $\frac{1}{O(nmw_{\max})}$, so that the overall probability of success will be at least $\frac{1}{2}$. Therefore we must scale up the weights by an additional factor of $nmw_{\max}$, and the entire algorithm will require $O(\log^2 n)$ time and $O(n^{2.5}n^2(nmw_{\max})^3) = O(n^{7.5}m^3w_{\max}^3)$ processors.

This processor bound can be reduced at the expense of a logarithmic factor in running time if we note that the values of $\pi(v)$ for a subset of the vertices of $\tilde{G}$ determine $\pi(v)$ for all the vertices of $\tilde{G}$. We will show that $O(\log \max(n, w_{\max}))$ rounds of finding $O(m)$ S^v-joins will suffice to determine all the $\pi(v)$.

To prove this observe that if, in the construction of $\tilde{G}$ from G, an edge pq was expanded into a path $pv_1v_2\ldots v_kq$, then a minimum S^{v_i}-join will either include the entire path from p to v_i or from v_i to q. Further, there is a unique vertex v_l on the path such that for $j \leq l$

the edges of the path from p to v_j are in the minimum S^{v_j}-join (and thus the path from v_j to q is not), and for $j > l$ the path from v_j to q is in the minimum S^{v_j}-join (and the path from p to v_j is not). This vertex v_l can be found by binary search on the vertices of the path from p to q. Further, we can carry out the binary search for each expanded edge of G in parallel. Each such path is of length $O(nmw_{\max})$; therefore, executing the binary search requires $O(\log \max(n, w_{\max}))$ rounds of finding $O(m)$ S^v-joins. We have replaced one round of finding $O(nmw_{\max})$ S^v-joins with $O(\log \max(n, w_{\max}))$ rounds of finding $O(m)$. This algorithm will be a factor of $O(\log \max(n, w_{\max}))$ slower and require $\Omega(nw_{\max})$ fewer processors.

Thus we have proved the following theorem.

Theorem 6.4.1 There exists a Las Vegas $\mathcal{RNC}$ algorithm for unary-weighted perfect matching that requires $O(\log^2 n)$ time and $O(n^{7.5}(mw_{\max})^3)$ processors on a CRCW PRAM, and a Las Vegas $\mathcal{RNC}$ algorithm that requires $O(\log^2 n \log \max(n, w_{\max}))$ time and $O(n^{6.5}m^3w^2_{\max})$ processors.

Corollary 6.4.2 There exist Las Vegas $\mathcal{RNC}$ algorithms for the following problems with the indicated time and processor requirements on a CRCW PRAM.

1. $O(\log^2 n)$ time and $O(n^{7.5}m)$ processors:
   - Finding a minimum T-join in an arbitrary graph.
   - Finding a maximum cardinality 1-packing of T-cuts in a bipartite graph.
   - Finding the T-join structure of an arbitrary graph.
2. $O(\log^2 n)$ time and $O(n^{5.5}(mw_{\max})^3)$ processors or $O(\log^2 n \log \max(n, w_{\max}))$ time and $O(n^{5.5}m^2w^3_{\max})$ processors:
   - Finding a minimum weight T-join when the edge weights are given in unary

Proof: Immediate using the arguments given above. ■

Note that certifying that a perfect matching in a bipartite graph is of minimum weight is much simpler, even when the edge weights are given in binary. It is well known that optimal dual variables for the minimum weight perfect matching problem can be found via a shortest paths computation, which can be carried out in $\mathcal{NC}$ [47, 118]. Therefore, given a candidate minimum weight perfect matching, we can attempt to find optimal dual variables via this shortest paths computation. If we fail, we know that the candidate perfect matching is not of minimum weight. This approach does not seem fruitful in the general non-bipartite case; finding dual variables does not appear to be any easier than finding the minimum weight perfect matching itself [27].

6.5 Planar Multicommodity Flow

We define the multicommodity flow problem as follows. Let $G = (V, E)$ be an undirected graph, with a set of *demand* edges $F \subseteq E$ with demand $q(f)$ associated with each edge $f \in F$. Let each non-demand edge have an associated capacity $w(e)$, and specify that w and q are integral. When does there exist for each $f \in F$ a flow function ϕ_f in $(V, E-F)$ between the two endpoints of f and of value $q(f)$ such that $\forall e \in E - F$

$$\sum_{f \in F} |\phi_f(e)| \leq w(e)?$$

Such a set of flow functions is called a multicommodity flow.

Construct the graph G^* by replacing $f \in F$ with $2q(f)$ parallel edges and $e \in E - F$ with $2w(e)$ parallel edges. Denote the image of F under this transformation as F^*. Seymour proved [111] that if a multicommodity flow exists in G, it can be constructed from a maximum packing of T-cuts in the (bipartite) planar dual of G^*, where T is the set of nodes in the dual of G^* that are adjacent to an odd number of edges of F^*. An easy consequence of this construction is that the resulting flow values are half-integral.

Since we have demonstrated an $\mathcal{RNC}$ Las Vegas algorithm for producing a maximum packing of T-cuts in a bipartite graph, this yields immediately a $\mathcal{RNC}$ Las Vegas algorithm for planar multicommodity flow when the demands and capacities are given in unary.

Chapter 7

Parallel Algorithms for the Assignment Problem[1]

7.1 Introduction

In this chapter we move to a more practical perspective on parallel algorithms for network problems. We present a computational comparison of five different implementations of parallel algorithms to solve the assignment problem, the problem of finding a minimum-weight perfect matching in a bipartite graph. We focus on solving the problem in a *dense* graph, where there is an edge between every two nodes. All of the algorithms are implemented on the Connection Machine CM-2, a massively parallel SIMD Computer manufactured by Thinking Machines Corporation. We also have attempted to evaluate other algorithmic approaches that we did not implement.

We have implemented three versions of Bertsekas' auction algorithm [9, 10, 13]: the standard Jacobi and Gauss-Seidel versions and a hybrid combination of the two that seems not to have been considered before. This implementation is particularly interesting since it uses two different levels of the potential parallelism of the Connection Machine. We have also implemented two versions of the method of multipliers [57, 98, 32, 31]. We compared these implementations on dense assignment problems with costs generated uniformly and randomly. Of the five algorithms

[1]This chapter describes joint work with Stavros Zenios [123, 122].

the hybrid algorithm proved definitively to be the best in these tests, and was, on 1000×1000 problems, an average factor of $5 - 10$ faster in Connection Machine time than the Jacobi code, the previous best algorithm implemented on the machine. This comparisons were on a 16,384 processor Connection Machine. The factor of improvement depended on the cost range and increases with problem size. The computational results on a 32,768 processor machine were also competitive with the best results achieved by other researchers on MIMD machines on a similar distribution of problem instances.

In the computational studies we have seen of algorithms for dense assignment problems the problem instances have been generated with costs chosen uniformly and randomly. We discuss the implications of such test data and compare our implementations on other sorts of distributions as well.

Motivation for this Study: The principal/initial motivation for this study is as follows. Goldberg implemented the Goldberg-Tarjan maximum flow algorithm on the Connection Machine CM-1, and tested it thoroughly. This implementation yielded excellent results and excellent parallel speedups [45]. However Goldberg points out that

> ... many maximum-flow problems that appear in practice are much smaller and much simpler than the examples we have generated, and most network-flow algorithms produce satisfactory results.... we have seen, however, that minimum-cost flow problems can be solved by iterating a generalized version of the maximum-flow algorithm. Since large and hard instances of the minimum-cost flow problem do appear in practice, fast maximum-flow algorithms are very important in this context [45].

We initially implemented and studied the algorithm to which Goldberg refers on the Connection Machine CM-2. The algorithm is due to Goldberg and Tarjan and computes a minimum-cost flow by successive approximations, where each approximation is computed by an algorithm that is quite similar to the maximum-flow code. However, despite the similarity between the **improve-approximation** routine of the minimum-cost flow algorithm and the Goldberg-Tarjan maximum-flow algorithm, the Connection Machine version of this minimum-cost flow algorithm performed quite poorly on networks generated by the standard random-network generator of Kempka and Kennington [121]. The basic problem is that, at least on these instances, the algorithm does not achieve good parallelism and leaves most of the machine idle most of the time. This phenomenon arose in several combinatorial applications on the machine, including shortest paths algorithms [55] and traveling salesman problem heuristics [96]. It also arose in an initial implementation of Bertsekas' Jacobi auction algorithm for the dense assignment problem. The auction algorithm and the Goldberg-Tarjan minimum cost-flow algorithm discussed above are quite similar, especially since the assignment problem is just a special case of minimum-cost flow.

We therefore decided to study alternative approaches to an auction-like strategy in order to try to understand how processor utilization might be increased in combinatorial algorithms on the Connection Machine. We focused on the assignment problem since it was a special case of the minimum-cost flow problem that nonetheless captured many of the same issues with regard to parallel algorithms. We focused on the dense problem since it yielded a very communication-efficient representation on the Connection Machine, whereas communication for sparser problems is comparatively slow. Furthermore, almost all of the other studies on parallel implementations of algorithms for the assignment problem concentrate on the dense problem, and thus we can use the results of this paper as benchmarks against the work reported by others. Finally, we believe that many of the techniques that we developed will prove useful in developing better implementations of algorithms for the sparse assignment problem and both sparse and dense minimum-cost flow problems.

The implementations of the methods of multipliers were motivated by the success of Eckstein in using these methods on sparse assignment problems [32, 31]. These methods also have tremendous potential to take advantage of the massive parallelism of the Connection Machine; however, they proved to be ineffective on dense problems. The discussion of these methods and the evaluation of other possible algorithms are included in an attempt to give a clearer picture of the possibilities of developing an algorithm superior to our hybrid algorithm.

The rest of this chapter is organized as follows. In Section 7.2 we describe other work on implementations of parallel algorithms for the assignment problem. In Section 7.3 we describe the relevant details about the algorithms we implemented, and in Section 7.4 we describe the implementations on the CM-2. In Section 7.5 we describe our computational results on the problems with costs chosen randomly from a uniform distribution on integers, and in Section 7.6 we describe the results on other distributions. In Section 7.7 we discuss the possibilities of other approaches to the problem and we give our conclusions in Section 7.8.

7.2 Previous Studies

A number of researchers have studied parallel solutions to large dense assignment problems on smaller scale MIMD parallel machines. These studies have concentrated entirely on the auction algorithm of Bertsekas and the Shortest Augmenting Paths Approach [67]. They have also concentrated entirely on costs that are generated randomly and uniformly.

Kennington and Wang [75] developed a parallel version of the shortest augmenting path (SAP) code of Jonker and Volgenant [67], which they tested on the Symmetry S81, with up to ten Intel 80386 cpus. They report solutions to dense 1200×1200 assignment problems with cost range $[0 - 1000]$ in approximately 15 seconds and cost range $[0 - 10000]$ in an average of under 20 seconds. They also report that the auction algorithm did not achieve results comparable to the shortest augmenting path in a serial implementation, and hence it was not parallelized. Zaki [125] continued this study on an Alliant FX/8, parallelizing and vectorizing both algorithms. He confirmed the observations of Kennington and Wang, for certain problem categories. However, his results show that the auction algorithm achieves much better speedups than the SAP code and also vectorizes very well. As a result, the auction algorithm outperforms SAP by a large margin when implemented on a vector/parallel architecture. He reports solutions of 2000×2000 problems with cost range [0-10000] in approximately 30 seconds with the auction algorithm, and 2 minutes with SAP.

Kempka, Kennington and Zaki [74] tested the auction algorithm on the Alliant FX/8 without the ϵ-scaling that typically makes the algorithm computationally effective and more stable. They report, nonetheless, solutions to a 1000×1000 dense problem with cost range $[0, 100]$ in under one second and a 4000×4000 problem in just over a half minute. Since they do not use scaling, their results are unpredictable: a 1000×1000 problem takes 12 seconds for cost range $[0, 1000]$, while a 2000×2000 problem with the same cost range takes over 255 seconds. For cost range 10000 they achieve an average of 33 seconds for 2000×2000 problems.

Balas, Miller, Pekny and Toth [4] implemented a parallel shortest augmenting path algorithm on the Butterfly Plus computer, with 14 processors. They were able to solve dense 1000×1000 problems with cost range [0 - 1000] in an average of 9.39 seconds, and cost range [0 – 10000] in 11.70 seconds. They also solved dense 2000×2000 problems with cost range [0 – 10000] in 30 seconds, 3000×3000 in a minute, and a dense 30000×30000 problem with 900 million variables in less than an hour.

Bertsekas and Castanon [11] did an extensive study of several variants of the algorithm on 20% dense problems on the Encore Multimax. They tested both Jacobi and Gauss-Seidel versions and a block-Jacobi implementation. An interesting feature of this study is that they develop asynchronous, as well as synchronous, parallel implementations. The asynchronous algorithm has a substantial advantage over its synchronous counterpart. They were able to solve problems of size 1000×1000 in under 10 seconds.

Castanon, Smith and Wilson [19] studied the effectiveness of different synchronous implementations of the Gauss-Seidel auction algorithm and the shortest augmenting paths code of Jonker and Volgenant [67] for solving both dense and sparse assignment problems on a variety of architectures. They demonstrated speedups of up to 60 for the Gauss-Seidel implementation of the auction algorithm for problems of size 1000×1000.

There are two studies of Connection Machine algorithms that are relevant to our work. Cindy Phillips wrote the first version of the Jacobi auction algorithm on the Connection Machine [95, 96]. Most of the important details of the Jacobi implementation were developed by her, however she was not able to thoroughly test the code and solves only one example for most cost ranges.

Eckstein did an empirical study of his alternating direction method of multipliers for general linear cost sparse networks on the Connection Machine CM-2 [32, 31]. The computational results were encouraging for sparse assignment problems.

There has been a variety of recent related work on Connection Machine algorithms for network optimization. We have already mentioned that Goldberg implemented a fast maximum flow algorithm [45]. Zenios and Lasken [127] solved nonlinear network problems; Zenios [126] developed algorithms for multicommodity transportation problems; Nielsen and Zenios [128] developed algorithms for stochastic network optimization models arising in financial applications. Eckstein [32, 31] has extensively studied the alternating step method for transportation problems. As stated above, his results for linear cost networks are not encouraging, but the results for sparse quadratic transportation problems are quite good and are competitive with the massively parallel row-action algorithm of Zenios and Censor [20]. Furthermore, his method appears to be the able to solve problems with mixed linear and quadratic objective terms with little additional difficulty.

7.3 Algorithms for the Assignment Problem

The assignment problem is to find the minimum-weight perfect matching in a bipartite graph. At times in this chapter we will phrase the problem in terms of n people and n objects, with a benefit a_{ij} associated with the assignment of object i to person j. We seek an assignment A of objects to people ($A(i) = j$ means that object i is assigned to person j) so that $\sum_{A(i)=j} a_{ij}$ is maximized. In a globally optimal solution any given person may not be assigned to his most valuable object. However, for a globally optimal assignment it is possible to assign a price π_i to each object i, so that if each person j views the profit associated with object i as $a_{ij} - \pi_i$ then each person is assigned to his most *profitable* object. This fact can be understood as a consequence of linear programming duality.

7.3.1 The Auction Algorithm

The auction algorithm finds the optimum assignment by finding such prices for all the objects. It produces an assignment A and prices π_i such that

$$a_{iA(i)} - \pi_{(i)} \geq \max_{j=1,\dots,n} (a_{jA(i)} - \pi_j).$$

The algorithm starts with each object assigned an arbitrary price; prices are adjusted upwards as people bid for their most profitable object. Each iteration of the algorithm consists of one or several currently unassigned people choosing the object that is most profitable to them and submitting a "bid" on the object. Each object that has been bid upon is assigned to the highest bidder, adjusts its price to the bid, and deassigns the person to whom it was previously assigned (if anyone).

Formally, each person bidding calculates the profits p_{best} and p_{next} associated with his two most profitable objects, and then bids $\pi_i + p_{best} - p_{next}$ on his best object i. A bid-upon object then is assigned to the highest bidder and sets its price to that bid.

Epsilon Scaling

Epsilon-scaling is used in the auction algorithm in order to improve upon the worst-case time bounds and computational behavior. We relax the condition that an object bids upon and is assigned to its favorite object:

$$a_{iA(i)} - \pi_i \geq \max_{j=1,\ldots,n} (a_{jA(i)} - \pi_j).$$

Instead we merely require that

$$a_{iA(i)} - \pi_i \geq \max_{j=1,\ldots,n} (a_{jA(i)} - \pi_j) - \epsilon.$$

Such an assignment is called *ϵ-optimal.* The algorithm runs in a series of *phases*, each phase taking an ϵ-optimal assignment and returning a set of prices and an assignment that is $\epsilon' = (\epsilon/c)$-optimal, for some user-specified constant c. Bertsekas proved that if an assignment is ϵ-optimal with $\epsilon < \frac{1}{n}$, and the a_{ij} are all integers, then the assignment is globally optimal [9]. Define $M_a = \max_{ij} a_{ij}$. Since *any* assignment is M_a-optimal, we can start with $\epsilon = M_a$ and in $O(\log(M_a n))$ phases we will produce an assignment that is $(\frac{1}{n+1})$-optimal and thus globally optimal. Each phase of the algorithm is a mini-auction as described in the previous section, except that instead of bidding $p_{best} - p_{next}$, a person can bid $p_{best} - p_{next} + \epsilon$. Each phase produces a ϵ-optimal assignment, and by successively lowering ϵ in this fashion we obtain an optimal assignment. The worst-case sequential complexity of the algorithm on dense problems is $O(n^3 \times \log(M_a n))$, and there is no known proof of worst-case parallel speedup. A summary of the algorithm is as follows.

Step 0 (Initialization) $\epsilon \leftarrow \max_{ij} a_{ij}$, $\pi_j \leftarrow 0$.

Step 1 Auction:

1.1 Some subset of the set of unassigned people determine and bid on their favorite object.

1.2 Each bid-upon object determines its highest bidder, raises its price to that bid, and assigns itself to that person, deassigning the person to whom it was previously assigned (if anyone).

1.3 If any person is unassigned goto 1.1.

Step 2 If $\epsilon < \frac{1}{n}$ **Stop**; else $\epsilon \leftarrow \epsilon/2$. People whose assignments are no longer ϵ-optimal deassign themselves.

Step 3 Goto Step 1.

Jacobi vs. Gauss-Seidel

Note that in the most general form of the algorithm any subset of those people unassigned can bid simultaneously. The two traditional parallel variants of the auction algorithm are the Jacobi version and the Gauss-Seidel version. By the Jacobi version we refer to an algorithm in which *all* unassigned people bid simultaneously on their favorite objects before the prices are adjusted, whereas in Gauss-Seidel only one person bids at a time. Since in the Gauss-Seidel version each bid takes advantage of the updated price information of the previous bids, it usually takes fewer total bids to produce an optimal assignment. The Jacobi method, however, has greater potential for a massively parallel implementation.

In general the terms "Jacobi" and "Gauss-Seidel" are not used solely with regard to the auction algorithm. "Jacobi" is used to refer to a method where the prices (dual variables) at time $t+1$ are updated only with respect to the information at time t, whereas a "Gauss-Seidel" iteration updates a dual variable with respect to the most recent information. Thus a Jacobi method allows the updating of all prices in parallel, whereas a Gauss-Seidel method allows two prices to be updated in parallel only if the update of one does not depend on the relevant values of the other. This Jacobi/Gauss-Seidel distinction is one of the primary differences between the two methods of multipliers we discuss.

7.3.2 The Method of Multipliers Algorithm

The *method of multipliers* (MOM) is a general method for a variety of problems in convex programming [57, 98]; we summarize here a specialization of the method to assignment problems suggested in [12] and we refer the reader there for a full development and explanation of the algorithm. Since the methods of multipliers we describe here are specializations of multiplier methods for linear programs, we first formulate the assignment problem as a linear program, and without loss of generality as a minimization problem. We let E denote the set of edges of the network; in our dense case E contains an edge (i, j) between every person i and object j.

$$\text{minimize} \sum_{\{(i,j)\in E\}} a_{ij} f_{ij}$$

$$\text{subject to}$$

$$\sum_{\{j|(i,j)\in E\}} f_{ij} = 1 \; \forall i = 1, \cdots n,$$

$$\sum_{\{i|(i,j)\in E\}} f_{ij} = 1 \; \forall j = 1, \cdots n,$$

$$0 \le f_{ij} \le 1 \quad \forall (i,j) \in A.$$

We assign dual prices r_i and p_j to the equality constraints. The method of multipliers for this problem results in a Gauss-Seidel iteration to minimize the Augmented Lagrangian. the iterative step has the form

Step 0 $f_{ij} \leftarrow 0, r_i \leftarrow 0, p_j \leftarrow 0.$

Step 1 Update the values of f as follows

$$f_{ij} = \left[f_{ij} + \frac{1}{2c(t)} [r_i(t) + p_j(t) - a_{ij} + c(t)(y_i(t) + w_j(t))] \right]^+, \forall \text{edges}(i,j)$$

where $y_i(t)$ and $w_j(t)$ are given in terms of $f_{ij}(t)$ by

$$y_i(t) = (1 - \sum_{\{j|(i,j)\in A\}} f_{ij}(t)) \; \forall i = 1, \cdots, n,$$

$$w_j(t) = (1 - \sum_{\{i|(i,j)\in A\}} f_{ij}(t))\ \forall j = 1, \cdots, n,$$

$c(t)$ is a nondecreasing sequence of of positive constants and $[x]^+$ indicates the projection of x onto $[0, 1]$.

Step 2 At the end of the minimization yielding $f_{ij}(t+1)$, $y_i(t+1)$ and $w_j(t+1)$ for all i, j, update the prices r_i and p_j according to

$$r_i(t+1) = r_i(t) + c(t)y_i(t+1), \forall i = 1, \cdots, n,$$

$$p_j(t+1) = p_j(t) + c(t)w_j(t+1), \forall j = 1, \cdots, n$$

Step 3 Check for convergence; if iteration has not converged goto Step 1.

7.3.3 The Alternating Direction Method of Multipliers

The *alternating direction method of multipliers*(ADMOM), due to Eckstein [32], takes a Jacobi approach to updating the augmented Lagrangian This method as well has a variety of applications to convex programming, [12, 32, 31, 33, 34]. The application of this method to the assignment problem results in a Jacobi-type algorithm which is more suitable for massively parallel computation than the method of multipliers. The iteration proceeds as follows.

Step 0 $f_{ij} \leftarrow 0, r_i \leftarrow 0, p_j \leftarrow 0.$

Step 1 Update the f_{ij} as follows

$$f_{ij}(t+1) = \left[f_{ij}(t) + \frac{1}{2c}[r_i(t) + p_j(t) - a_{ij} + c(y_i(t) + w_j(t))]\right]^+ \forall \text{edges}(i, j),$$

$$r_i(t+1) = r_i(t) + cy_i(t+1), \forall i = 1, \cdots, n,$$

$$p_j(t+1) = p_j(t) + cw_j(t+1), \forall j = 1, \cdots, n$$

where $y_i(t)$ and $w_j(t)$ are given in terms of $f_{ij}(t)$ by

$$y_i(t) = \frac{1}{n}(1 - \sum_{\{j|(i,j)\in E\}} f_{ij}(t)), \forall i = 1, \cdots n,$$
$$w_j(t) = \frac{1}{n}(1 - \sum_{\{i|(i,j)\in E\}} f_{ij}(t)), \forall j = 1, \cdots n,$$

and c is a constant.

Step 2 Check for convergence; if iteration has not converged goto Step 1.

Note that in ADMOM all f_{ij} can be updated simultaneously while MOM relies on y and w being up to date, and therefore only one value f_{ij} can be updated in each row and column at a time. Therefore we can only do n updates at a time for MOM while we can do n^2 for ADMOM; however, due to the Gauss-Seidel nature of the minimizing iteration in MOM, we would expect that the algorithm would converge much more quickly. Furthermore the price updates for ADMOM are divided by the number of arcs incident to that node, which for dense problems is n. Therefore for large dense problems the prices will only change a small amount in each iteration and therefore the number of iterations has the potential to be large. Note that neither of these algorithms has been proved to be a polynomial-time algorithm, although they have been proved to converge to the correct answer in a finite amount of time.

7.4 Designs for Massively Parallel Implementation

In this section we discuss the issues involved in implementing efficiently the algorithms of the previous section on a Connection Machine CM–2.

7.4.1 The Connection Machine CM–2

The Connection Machine CM–2 is a massively parallel computer with up to 65,536 processors. Each processor has a single-bit processing unit and 64K or 256K bits of local RAM. The processors run in SIMD mode and are connected in an n-cube topology. The system software provides global maximum operations as well as *scan* and *spread* operations that are parallel prefix operations [14]. The CM–2 uses a front end such as a SUN-4, VAX or Symbolics Lisp Machine. Parallel extensions to the programming languages LISP, C and FORTRAN, via the front-end, allow the user to program the Connection Machine and the front-end system. For further information see [26] and [58].

A Connection Machine can emulate a large number of processors by having each physical processor simulate a number of *virtual* processors. The ratio of the number of virtual processors to the number of physical processors is referred to as the *virtual processor ratio*, or *vp-ratio.* Using standard Gray coding the processors of the CM can be configured as a k-dimensional grid; to represent a $n \times n$ assignment problem we configure the CM as a $N \times N$ grid, where N is n rounded up to the nearest power of two. Row i is associated with object i and column j is associated with object j. In particular, processor (i, j) stores the value a_{ij} of object i to person j, local variables applicable to person j such as the most profitable object to that person, and local variables applicable to object i, such as its price. A specified number of virtual processors, along with a configuration of the machine, is called a *vp-set*, and it is possible to use several different vp-sets in the same computation and to switch between them when desired.

A mapping of virtual processors in a grid to the physical processors of the CM is known as a *geometry*. The user has the freedom to dictate a variety of the features of this mapping. In particular we exploit the ability to specify which axes or directions of the grid should have more physical processors representing them, and which should have more virtual processors. In other words, we can specify that virtual processors that are adjacent along one axis are on different physical processors, and that processors that are adjacent on another axis are on the same physical processor. The mapping of the $N \times N$ grid onto the physical CM will differ among our implementations, and we will use alternate representations as well, but this is the basic representation common to all the codes.

7.4.2 Massively Parallel Implementations of the Auction Algorithm

In this section we discuss the implementation of the Jacobi and Gauss-Seidel variants of the auction algorithm, together with a new version that we refer to as a "hybrid" algorithm.

The Jacobi Algorithm

In the implementation of the Jacobi[2] algorithm we have one $N \times N$ vp-set which is used both to store the data and for computation. It is mapped onto the physical CM processors in the default fashion, which balances the two axes so that they have a comparable physical processor/virtual processor makeup. A detailed summary of the Jacobi implementation is as follows.

Step 0 We keep a copy of ϵ in each processor. Using a global maximum operation and a broadcast, set $\epsilon \leftarrow (n+1)M_a$. In each processor set $\pi \leftarrow 0$. We keep two boolean variables in each processor: **assigned-here**, which is True in processor (i,j) if object i is assigned to person j, and **person-assigned**, which is True in all processors of column j if person j is assigned to an object. In all processors set **person-assigned** $\leftarrow$ False and **assigned-here** $\leftarrow$ False. Also scale all values of a by $n+1$.

Step 1 Determine if everyone is assigned (a global or operation of **person-assigned**). If a person is unassigned proceed to Step 2. If every person is assigned to an object and $\epsilon < 1$ the algorithm terminates. If $\epsilon > 1$ reduce its value, deassign everyone whose current assignment is no longer ϵ-optimal for the new ϵ, and proceed to Step 2.

Step 2 Select all the processors associated with unassigned people. Set **profit** $\leftarrow a - \pi$. Within each column j find the row index i_j of the most profitable object by forming

[2]Preliminary experiences with this algorithm have already been reported on in [95]; it is included here for completeness and in order to present additional computational results. The description of the algorithm is adapted from that paper.

Number of Bidders	Fraction of Iterations
1	38.60%
1-10	82.35%
1-100	95.95%
1-500	98.78%

Table 7.1: Statistics on Number of Bidders active

the concatenation of the profit and column number j in each processor and doing a grid spread-with-max. Set `best` $\leftarrow i_j$. Turning off processor (`best`, j) do another grid spread-with-max to find the profit p of the next-best object and set `next-best` $\leftarrow p$. Person j bids on object `best` by setting the variable `bid` in processor (`best`, j). The value of the bid is computed as follows: Let $w_{\texttt{best},j}$ be the maximum profit from all objects except `best`. The bid from j to `best` is $a_{\texttt{best},j} - w_{\texttt{best},j} + \epsilon$.

Step 3 Using a grid spread-with-max along the rows, determine the maximum bid on each object and update the prices π within the columns. For all objects bid upon, assign the object to the highest bidder by setting `assigned-here`. Update `person-assigned` using or-grid-scans within the columns. Go to Step 1.

The CM has the ability to potentially perform thousands of bids at once; therefore, this seems to be a very attractive method for this architecture. However, this approach leads to a large sequential tail. Most people are assigned to objects very quickly, and the bulk of the computational time is spent in establishing the last few assignments, hence greatly reducing the amount of parallelism. Table 7.1 gives a typical breakdown of processor utilization encountered in a 1000×1000 problem with cost range $[0 - 1000]$.

Two optimizations were implemented to minimize this tail effect. The first was to optimize the case when only one bidder was active: the preferred object for that sole bidder was found by a global maximum over all the processors as opposed to the more powerful but slower max-scan operation. The latter was unnecessary for the case of one bidder.

The second optimization was the truncation of the tail: instead of running each phase until a complete ϵ-optimal assignment was achieved, the auction was terminated when $k\%$ of the

Problem Size	Maximum Cost	Speedup(its.)	Speedup(Time)
128	100	5.62	4.61
128	1000	4.30	3.70
256	100	3.23	3.28
256	1000	7.37	6.35

Table 7.2: Improvements in iterations and time gained by truncation of the tail. Each entry is an average over five randomly generated examples.

people have been matched. k increases as *epsilon* decreases, so that $k = 100$ in the last phase. This optimization resulted in substantial speedups. An implementation that initially matches only 80%, and matches progressively more in successive phases does an average of 5.1 times fewer iterations than an implementation that completes each phase, on randomly uniformly generated problems of size 128-256. (See Table 7.2.)

The Gauss-Seidel Algorithm

In this version, where we do one bid at a time, we must continue to store the $n \times n$ problem on the CM, which will necessarily be configured at a high vp-ratio. We would like, however, to perform the computations at a lower vp-ratio, and therefore more quickly, on a grid of reduced dimension. We introduce a mapping of the $N \times N$ grid to the physical machine that will enable us to extract the information necessary for one bid, and compute the bid in a vp-set with vp-ratio 1. We do this in a fashion that requires *no communication* between the two vp-sets.

We define the geometry of the $N \times N$ vp-set, which we will call **data-vp-set**, so that the y axis is physical. Processors (i, j) and (i, k) are on different physical processors for all $j \neq k$. All of the virtual bits are along the x axis; processors $(0, j), (1, j), \ldots, (\text{vp-ratio} - 1, j)$ will all be mapped to the same physical processor, as will all processors (q, j), where $q \in [\text{vp-ratio} * k, \text{vp-ratio} * (k+1) - 1]$. We define a second vp-set, **compute-vp-set**, of vp-ratio 1 that has N rows and (Number of Physical CM processors)/N columns. For example, if $N = 1024$ and we are working on a Connection Machine with 16,384 processors, **compute-vp-set** will be a 1024×16 vp-set (1024 rows, 16 columns). A processor in **compute-vp-set** shares the same physical processor with 64 processors in the **data-vp-set**. Note that an entire column in **data-vp-set** shares the same

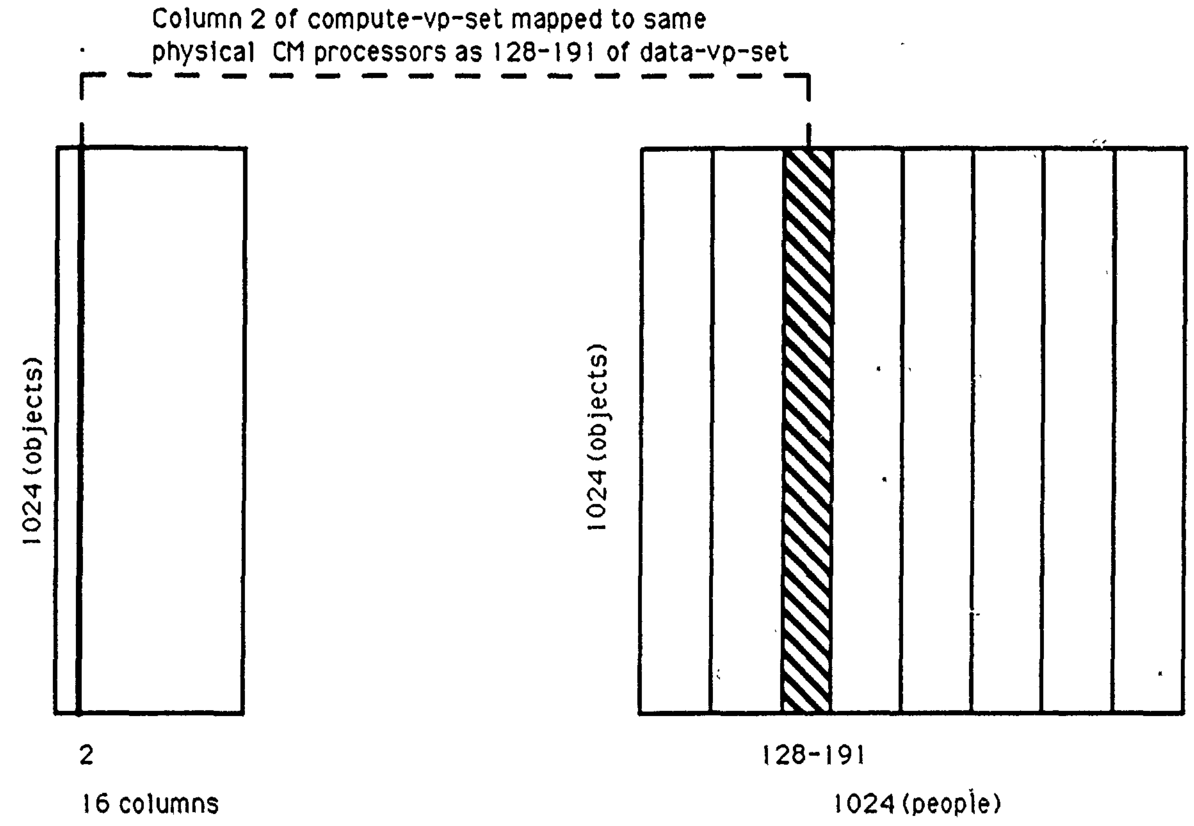
Column 2 of compute-vp-set mapped to same
physical CM processors as 128-191 of data-vp-set
1024 (objects)
1024 (objects)
2
16 columns
128-191
1024 (people)

Figure 7.1: The two vp-sets for the Gauss-Seidel algorithm, compute-vp-set and data-vp-set for a 1024×1024 problem on a 16K CM. Each vertical block of data-vp-set represents 64 virtual columns that all reside on the same physical column. Only 8 blocks are portrayed here.

physical processors with one of the columns of compute-vp-set. We will transfer the information we need for one bid from data-vp-set to compute-vp-set by having a processor in compute-vp-set just point to the relevant data in data-vp-set. See Figure 7.1.

To execute the bid for the ith person we select his column c_{data} in data-vp-set, and identify the column in compute-vp-set with which it is coincident: $c_{compute} = \lfloor \frac{c_{data}}{vp-ratio} \rfloor$. Each parallel variable has a pointer to the physical location where its data resides; we simply change the pointer of the parallel variable in compute-vp-set to point at the data in the data-vp-set. For example, in our 1024×1024 problem on a $16,384$ processor machine, suppose person 303 is bidding. All the relevant information resides on the same physical processors as does column 4 of the compute vp-set. Pointers are changed so that column 4 points to the data of column 303 in **data-vp-set**, and the bid is carried out. Note that the price information need only be stored in **compute-vp-set**. The Gauss-Seidel algorithm is thus as follows:

Step 1 Pick an unassigned person and calculate with which physical column he is associated; set the appropriate pvars in **compute-vp-set** to point to his values. (Need to change only one pointer per pvar.)

Step 2 In **compute-vp-set**, calculate his bid. (This requires two global maximum computations: one to calculate the best object and one for the next best.)

Step 3 Update the price information, which only need be kept in **compute-vp-set**.

Step 4 Goto step 1.

In contrast to the Jacobi implementation, where all information is kept on the CM, it is more efficient here to keep track of who is assigned, and to what object, in front-end lists and arrays. This avoids significant amounts of communication. In fact we utilize a copy of the a_{ij} that is kept on the front end as well to calculate the bid and in this way avoid CM to front-end communication time.

Note that from the end of one phase to the start of the next, when epsilon is decreased to ϵ', often many of the ϵ-optimal assignments from the previous phase are ϵ'-optimal as well, and need not be recomputed in the next phase. In the Jacobi code these assignments are identified and preserved for the next phase; this is easy with a Jacobi representation since each assignment can be checked in parallel. In the Gauss-Seidel case, when this information is stored on the front end, it is a sequential computation to determine who is ϵ'-optimal and thus we do not check. Not preserving these assignments does increase the total number of Gauss-Seidel bids, since we are throwing away information, but the gain in computation time outweighed the increase in iterations; thus we chose not to preserve them. We tested our Gauss-Seidel implementation against a sequential Gauss-Seidel implementation that we obtained from D.P. Bertsekas on problems of size 64 – 256 and found that the number of bids they performed was comparable.

The Hybrid Jacobi/Gauss-Seidel Algorithm

The Jacobi code is very efficient when a large number of people are unassigned and are bidding and the Gauss-Seidel code makes effective use of the machine when there are few active bidders. It can do one bid at a time very quickly and also decreases the total number of bids needed. These characterizations suggest that a better use of the CM is a hybrid combination of these approaches. Execute the Jacobi algorithm early in the phase when large numbers of people are unassigned. When most are assigned, switch to Gauss-Seidel. Note that the deterministic algorithm with the best worst-case running time bound $O^{*}(n^{\frac{5}{2}}\log(nM_a))$ uses a very similar two-phase approach with the second phase being a shortest paths algorithm [46]. Ahuja and Orlin devised a sequential two-phase auction algorithm with an improved running time over a standard vanilla auction algorithm [1].

Making the transition from Jacobi to Gauss-Seidel will require a certain amount of communication of the data during a phase, since the way that the grids are mapped onto the Connection Machine is different for Jacobi and Gauss-Seidel. This cost is far outweighed by the gains in computation time.

The hybrid algorithm uses two thresholds, *thresh*1 and *thresh*2. *thresh*1 determines in which phases we employ both methods; *thresh*2 determines when in the phase we switch to Gauss-Seidel. If a phase is aiming to assign $k\%$, $k\% > thresh1$ then we switch to Gauss-Seidel when $thresh2 \times k\%$ of the people are assigned. Computational testing showed that for problems of size 1000×1000 and 2000×2000 with uniformly randomly generated costs the best setting of these parameters, although different for different cost ranges, was $thresh1 = thresh2 = 97$ or 98. Interestingly, for both problems this is close to the $\sqrt{n}$ turning point used theoretically in [1].

7.4.3 Massively Parallel Implementations of the Methods of Multipliers

In contrast to the auction algorithm, which only has the *potential* to do n bids at a time, MOM *always* does n updates at a time and ADMOM does n^2. Therefore these methods are very attractive for parallel computation. The description of the Connection Machine implementations of these algorithms is relatively straightforward given our previous discussion. We configure the

Size	cost	Jacobi Bids	Jacobi Iterations	Average Parallelism	GS Bids
128	100	3837	147	26.49	1567
128	1000	4114	215	20.77	2099
256	100	10210	813	14.56	5102
256	1000	9054	264	40.56	6244

Table 7.3: Jacobi vs. Gauss-Seidel Statistics

Connection Machine as an $N \times N$ geometry, with equal preference given to the physical makeup of each axis, (as we did with the pure Jacobi auction algorithm). Each processor (i, j) stores the current value of f_{ij}, p_i, r_j, y_i, and w_j. For the ADMOM algorithm, we need merely to do one spread-with-add operation along each axis in order to calculate $\sum_{\{j|(i,j)\in E\}} f_{ij}(t), \forall i = 1, \ldots, n$, and $\sum_{\{i|(i,j)\in E\}} f_{ij}(t), \forall j = 1, \ldots, n$; then all that is required is several arithmetic operations that all happen within each processor with no further communication required. For the MOM algorithm we must select groups of n processors such that only one processor is selected in each row and column. The strategy that we use is to select processors (i, j) such that $i + j \equiv k (\text{mod } n)$, and loop over $k = 0, \ldots, n-1$.

7.5 Computational Results and Discussion

In this section we discuss the performance of the two algorithms on dense problems. All the problem data was generated by generating integer costs randomly and uniformly using the Connection Machine random number generator. This is a very specialized distribution and problems drawn from this distribution are generally understood to be easy, in that most of the lower weight edges are not necessary. In fact, one can usually solve these problems by removing 75% of the edges, those with smaller costs, and solving the remaining problem. If the result is not optimal it can usually be patched up quickly [92].

Despite these considerations, we believe that computational results on this distribution still are meaningful, for several reasons. First of all, all computational studies which we know of test only on this distribution; therefore testing such problem instances provides some ability to compare different implementations and architectures. Secondly, removing the "bottom" 75%

Size	cost	Jacobi Time	Gauss-Seidel Time	Hybrid Time
1024	100	197.5/106.2	585.5/124.8	105.1/25.5
1024	1000	414.2/219.5	242.7/46.5	58.6/22.1
1024	10000	276.7/146.0	222.2/44.3	92.3/20.8

Table 7.4: Running times of the algorithms, in seconds, averaged over five random examples. Both total and CM time are given.

of the arcs is not particularly useful on the Connection Machine. Although it would allow the machine to be configured at a lower vp-ratio, the resulting sparse communication pattern that would result would be significantly slower than that available when one has a fully dense problem [31]. Finally, despite the fact that these problems are understood to be easy, *they seem not to be easy at all for a massively parallel architecture* due to the tail phenomenon we have discussed. Their "easiness" may in some sense cause this difficulty, since most of the assignments are easy to compute and it is the computation of the last few that takes a great deal of time. Nonetheless, it is important to understand if a massively parallel architecture can achieve good results on these sorts of instances.

In the next section we discuss computational results on instances drawn from different distributions.

7.5.1 The Performance of the Auction Algorithm

The auction algorithms were initially implemented in a combination of *Lisp and Lisp/Paris and were run with a SUN4 front end. A comparison of the number of bids done by the Jacobi and Gauss-Seidel codes on problems of moderate size is given in Table 7.3. We see that the Gauss-Seidel code can do significantly fewer bids than Jacobi, by as much as a factor of two or greater. We also see that, as expected, the average number of people bidding in the Jacobi code is much less than n.

Table 7.4 gives data on the running times of the algorithms on fully dense problems of size $n = 1024$, on a 16,384 processor CM2. Each running time is given in seconds and is the average of five problems. The Hybrid algorithm is faster than both the Jacobi and Gauss-Seidel codes in all cost ranges; this was true as well for a number of smaller problem sizes. Further, this also held true for a variety of settings of the parameters of the code, such as factor by which we divide epsilon in each iteration, the initial percentage matched, etc. This led us to believe that the superiority of the hybrid approach was fairly robust with respect to modest modifications to the algorithms.

The parameter settings that proved to work best are 80% for the initial percent to be matched and a factor of 2 to divide epsilon, except for the Gauss-Seidel code where a factor of 4 seemed superior. In joint work with Vu Lephan that is reported on in [120], we discovered that the same parameter settings were the best for a sparse implementation of a Jacobi auction algorithm, over a wide range of problem sizes and degrees of sparsity.

It is interesting and not intuitive that the Gauss-Seidel code would often do as well or better than the Jacobi code, especially when one considers Connection Machine time alone. However, one must take into account the fact that the time of one Gauss-Seidel bid is $70-80$ times faster (in Connection machine time) than a Jacobi parallel bid on a 1024×1024 problem running on a $16K$ machine. The total times are significantly larger than the CM times. This partially reflects the strategy of letting the front end execute as much of the inherently sequential part of the problem as possible, but mostly reflects the Lisp front end code. Upon recoding in C/Paris this discrepancy basically disappeared.

Based on this testing we chose the hybrid algorithm as the most successful and recoded it in C/Paris. The C front end code runs much faster, and this led to significant improvements in the overall running times; the discrepancy between front end times and CM time became very small. In Tables 7.5 and 7.6 we give results on the performance of the algorithms on both 16K and 32K machines, for problems of size 1000-2000, over various cost ranges. Both the total number of iterations and the number of Jacobi iterations are reported. Each number is the average of ten randomly generated examples; we used different sets of examples for the 16K and 32K machine in order to give some idea of the variation possible in algorithm performance over two similar sets of problems of the same size and cost range.

The number of Jacobi iterations is surprisingly small, but during these iterations hundreds of bids are carried out at once. Another interesting fact is that for a specific problem size and cost range the number of Jacobi iterations is always about the same; for almost all size and cost ranges it was never more than five away from the average.

Size	cost	Time	Total Iterations	Jacobi Iterations
1000	100	34.6	16252	128
1000	1000	22.2	13637	119
1000	10000	15.4	5031	141
1000	100000	15.3	4923	144

Size	cost	Time	Total Iterations	Jacobi Iterations
1000	100	28.7	21850	126
1000	1000	17.5	10141	120
1000	10000	8.1	2000	141
1000	100000	9.8	3578	144

Table 7.5: Running times of the C/Paris hybrid auction algorithm on 1000×1000 problems. The top table is for a 16K machine, the bottom table for a 32K machine. Time is in seconds, averaged over ten random examples. 5 of the 20 examples with cost range $[1 - 100]$ tested ran for more than 100000 iterations, and their running times are not included.

We note that the cost range 100 for size 1000 problems is particularly difficult for our implementation; this is reflected both in the increased number of iterations and in the fact that on approximately 25% of the random instances we generated the code ran for more than 100,000 iterations, although it always terminated. After initial testing we ran each code only up to 100000 iterations and thus those examples of size 1000 and cost 100 that ran longer are not included in the averages.

We also note that given our vp-ratio 1 implementation of a Gaüss Seidel bid, the time for one bid is not dependent on problem size or machine size as long as n is smaller than the size of the machine. Our code averages 1000 Gaüss-Seidel bids per second on both a 16K and 32K CM–2. This of course is not the case for the Jacobi phase, e.g. one Jacobi parallel bid for a 1000×1000 problem on a 16K CM–2 takes .07 seconds, whereas on a 32K machine one parallel bid takes .04 seconds. Therefore, as bigger problems are solved and the vp-ratio increases, the factor by which the hybrid approach outperforms the Jacobi approach will increase. We were limited to 2000×2000 problems only because of memory constraints of the Connection Machine we were using. Currently Connection Machines exist with a factor of 16 more memory than the machine to which we had access; on such a machine with 16384 processors we would be able to solve 8000×8000 problems; with a fully-configured machine with $65,536$ processors we should

Size	cost	Time	Total Iterations	Jacobi Iterations
2000	100	60.2	2817	242
2000	1000	96.3	60936	148
2000	10000	60.4	23657	152
2000	100000	57.9	18004	156

Size	cost	Time	Total Iterations	Jacobi Iterations
2000	100	33.5	2368	241
2000	1000	50.3	22955	150
2000	10000	45.3	26739	151
2000	100000	36.3	15263	157

Table 7.6: Running times of the C/Paris hybrid auction algorithm on 2000 × 2000 problems. The top table is for a 16K machine, the bottom table for a 32K machine. Time is in seconds, averaged over ten random examples.

be able to solve a 16000 × 16000 problem.

7.5.2 The Performance of the Multiplier Methods

We began by testing the methods of multipliers on small problems in order to understand their behavior. The performance of MOM and ADMOM on randomly generated problems of size 64 × 64 and 256 × 256 is recorded in Tables 7.7, 7.8, and 7.9. In Table 7.8 the entry (x, y) means that ADMOM ran for an average of x iterations before converging, and that it converged within 10000 iterations for $y/10$ examples. We imposed an arbitrary cutoff here of 10000 iterations. An asterisk indicates that the methods never converged with that parameter setting on that cost range; an X means that we did not test that combination fully since initial testing indicated it would not converge. Based on our experience with ADMOM and some preliminary testing of MOM we chose a reduced set of c on which to carefully test MOM. Table 7.8 gives the results; here we imposed an larger arbitrary cutoff of 20000 iterations, since due to the Gauss-Seidel nature of the algorithm the number of iterations is expected to be larger. Ten random examples were considered for each cost range; again, * should be interpreted as meaning none of the random examples converged.

The tests on the size 64 problems indicate that indeed the number of minimizations of the

Cost	c=1	c=5	c=10	c=25	c=100	c=500
100	(1000,2)	(400,2)	(400,2)	(550,2)	(1400,2)	(5150,2)
1000	(8130,3)	(2880,10)	(1540,10)	(690,10)	(620,10)	(3450,8)
10000	*	*	(9800,1)	(6040,10)	(1740,10)	(730,10)
100000	X	X	X	X	(7750,10)	(2720,10)

Table 7.7: Performance of ADMOM on dense problems of size 64. each combination of cost and c was tested on 10 randomly generated problems. The entry (x, y) = average of x iterations on the $y/10$ problems that converged within 10000 iterations. A * indicates that none of the tested examples in that category converged within the limit, and an X indicates that that category was not tested fully since preliminary testing indicated it was sure not to converge.

augmented lagrangian is much smaller for MOM than for ADMOM, but the tests on size 256 problems, given in Table 7.9, indicate that total number of iterations of MOM gets unmanageable for size 256 problems, since one minimization requires 256 iterations of the inner loop. A further feature of MOM is that fairly frequently it returns non-integral but very close to optimal solutions; this behavior was not observed with ADMOM. The number (out of 10) of solutions that were integral is recorded in the Table 7.8 as well.

Both methods were highly sensitive to choice of c, and different values of c were preferable for different cost ranges.

Given this preliminary data we deemed it unnecessary to test MOM on 1000×1000 problems, due to its disappointing behavior on smaller ones. We did test ADMOM on dense problems of size 1000×1000, for c equal to each of $25, 100, 500, 1000$. We tested all 4 cost ranges with each c, on five randomly generated examples. None of the examples converged within 10000 iterations, which was approximately 2 minutes of time on a 32K CM–2. We thus conclude that for problems of this size MOM and ADMOM are inferior to a hybrid auction approach and in general not practical ways of solving large dense problems.

7.5.3 Comparisons With Other Codes on the Uniform Distribution

In Table 7.10 we compare our computational results with those reported by [4], [74] and [75] on uniformly randomly generated cost ranges. Our algorithm seems to be comparable or superior at cost ranges where the maximum cost is at least 10 times the problem size, and compares

Cost	c=25	# Integral	c=100	# Integral	c=500	# Integral
100	(9850,10)	0	(18430,1)	0	(13300,9)	0
1000	(5766,10)	7	(11460,9)	5	(16410,10)	1
10000	(7360,10)	10	(3827,10)	9	(5887,10)	4
100000	*	0	(12460,9)	9	(4530,10)	8

Table 7.8: Performance of MOM on dense problems of size 64 of various cost ranges.

Cost	c=25	c=25	c=100	c=100	c=500	c=500
1000	(26.5,2)	*	(58.5, 2)	*	(196,1)	*
10000	(86,7)	(181.7,2)	(39.9,7)	*	(67.8,8)	*
100000	*	*	*	*	(57.7,10)	*

Table 7.9: Data on 256×256 problems. We imposed an arbitrary cutoff of 20000 iterations. First column for each value is ADMOM, second is MOM.

particularly poorly in small cost ranges. To the best of our knowledge most of the computational studies on the auction algorithm started from a well-developed and tested code authored by Dimitri Bertsekas and Paul Tseng, and thus incorporates their experience in testing this algorithm. Due to the different architecture on which we were working it was not possible to directly adapt this code, and we were interested largely in the issues involved in creating an efficient massively parallel implementation of this algorithm. We believe that some of the heuristics developed by Bertsekas and Tseng could be incorporated into our code in modified form and potentially significantly improve the number of iterations required.

7.6 Other Input Distributions

We also tested our hybrid auction algorithm on dense problems with costs generated from a *Cauchy* distribution. For each edge we randomly and uniformly chose an integral $x \in [0, 1000]$, and set the cost of the edge to be $C(x)$, where

$$C(x) = \frac{b_1}{1 + \frac{(x-500)^2}{b_2}}.$$

Cost	Hybrid	Balas et. al	Kempka Kennington Zaki	Zaki	Kennington Wang
100	28.7	2.01	.758	.56	5.22
1000	17.5	9.39	12.083	12.49	8.27
10000	8.1	11.70	11.48	11.98	11.34
100000	9.8	–	–	–	13.61

Cost	Hybrid	Balas et. al	Kempka Kennington Zaki	Zaki	Kennington Wang
100	33.5	5.52	3.813	1.55	–
1000	50.3	23.20	255.56	257.2	–
10000	45.3	30.09	32.96	32.8	–
100000	36.3	–	–	–	–

Table 7.10: Comparison of the hybrid auction code on a 32K machine with other parallel codes. Times are given in seconds; the top table is $n = 1000$ and the bottom is $n = 2000$. We compare with the Jacobi auction code results of Kempka, Kennington and Zaki, the Gauss-Seidel auction code results of Zaki, and two SAP codes. These codes are discussed in the introduction. An entry with a –– indicates those researchers do not report results for that range.

We tested dense problems of size = 128, 256 and 512. For each of these we tested three different settings of the b_i: (1) $b_1 = 10000, b_2 = 500$, (2) $b_1 = 10000, b_2 = 100$ and (3) $b_1 = 1000, b_2 = 500$. We ran the hybrid algorithm with $thresh1 = thresh2 = \sqrt{n}$, and compared it to the Jacobi algorithm. For each setting of the b_i and each problem size we generated ten examples. We also ran the hybrid code on problems with uniformly generated costs in the same cost ranges, for the sake of comparison.

The results of these experiments are essentially identical to those from experiments on the uniform distribution. Since we ran these experiments on a 8192-processor Connection Machine, we ran somewhat smaller problems and can not compare our results directly, but (1) the running times of the hybrid algorithm on the Cauchy distribution were comparable to those on the uniform distribution, and (2) the qualitative behavior that led to the success of the hybrid algorithm, namely the large sequential tail, was observed as well. Average speedups observed on the 512×512 problems were in the range of $5 - 10$.

7.7 Other Possible Approaches

In this section we discuss several other possible approaches to designing Connection Machine algorithms for dense assignment problems.

Matching by Matrix Inversion

From a theoretical perspective, the fastest algorithm for the assignment problem is the randomized algorithm of Mulmuley, Vazirani and Vazirani [90]. This algorithm is quite simple conceptually, requiring only the calculation of a matrix inverse and a matrix determinant. Unfortunately, in order to guarantee with probability $\frac{1}{2}$ that the algorithm will succeed the edge weights must be scaled up by a factor of n^3. Since the entries of the matrix to be inverted are exponential in the edge weights, this computation is impossible for the cost ranges we consider.

When the costs of the edges are generated randomly and uniformly in a large enough cost range, the techniques of [90] indicate that even if the edge weights are not scaled up further the algorithm will succeed with high probability. Even the unscaled edge weights, however, lead to unmanageably large matrix entries. We experimented with computing only with the first digit or two of the edge weights; these computations yielded close approximations to the optimal assignment value but never in our experiments yielded the optimum value. Therefore we could find no way to convert the algorithm of [90] into a reasonable Connection Machine algorithm.

Shortest Augmenting Paths

In its most basic form the shortest augmenting paths algorithm solves the assignment problem by finding n shortest paths. Each of these shortest paths computations would require at least a few spread operations; several thousand spread operations would be much slower than a hybrid auction algorithm on a 1000×1000 or 2000×2000 problem.

A modified form of the shortest paths approach involves an initial auction-like phase that accomplishes a large number of assignments; the last assignments are established by shortest paths computations [67]. Such an approach has potential to be competitive with the hybrid auction algorithm; however, the fact remains that the shortest paths computations require spread operations at a high vp-ratio, in contrast to the hybrid algorith which processes the tail with global maximum operations at vp-ratio 1. Since these can be up to a factor of 100 faster than a spread for the problem and machine sizes we have discussed, it is very unlikely that a shortest-paths based approach can outperform the hybrid auction algorithm.

Network Simplex

It is the general consensus of the computational optimization community that the network simplex algorithm is inferior to both the auction algorithm and the shortest paths algorithm as a sequential algorithm for the assignment problem [75]. It may be possible to do one pivot quickly in parallel, in which case the performance of the algorithm depends on the number of iterations. To do one pivot quickly in parallel one would have to be able to quickly find the path between two vertices of a tree; the tree only changes by one arc from iteration to iteration. Doing $O(n)$ or even $O(\log n)$ spreads to find this path would be too slow, since spreads at high vp-ratios are slow in comparison. Therefore just using a standard path-finding algorithm would be ineffective. We do not see how to do this in a constant number of Connection Machine operations, although we have not studied the problem in great depth.

7.8 Conclusions

We have seen that for large dense problems the two methods of multipliers take good advantage of the massive parallelism of the Connection Machine, but are not computationally effective methods for dense problems due to the large number of iterations required for convergence. The auction algorithm is more difficult to implement efficiently on the Connection Machine, but we have presented several methods to achieve an implementation of a variant of this algorithm that is competitive with the best MIMD algorithms on large problems.

Massive Parallelism is best exploited algorithms which use little complex communication and must process a huge amount of data processed at every moment. Our experiences with the auction algorithm as a method to solve the assignment problem indicate that it does not fall into this paradigm. This is not an isolated phenomenon, but appears in a number of combinatorial approaches to optimization problems [121]. We have, however, established several methods that can bring combinatorial techniques closer to fast implementations on massively parallel architectures. It would seem that currently the best approach is to utilize massive parallelism while the problem continues to be massively parallel, and then to switch to another technique that is better suited to the tail of the problem. We have yet, however, to produce CM techniques for this problem that are significantly better thanthose implemented on smaller scale MIMD machines. It is a challenging open problem to more fully exploit massive parallelism in this field.

Bibliography

[1] R. K. Ahuja and J. B. Orlin. New scaling algorithms for the assignment and minimum cycle mean problems. Sloan Working Paper 2019-88, MIT, Cambridge, MA, 1988.

[2] R. J. Anderson and E. Mayr. A P-complete problem and approximations to it. Technical report, Stanford University, 1986.

[3] D. Applegate and B. Cook. A computational study of the job-shop scheduling problem. *ORSA Journal of Computing*, 3:149–156, 1991.

[4] E. Balas, D. Miller, J. Pekny, and P. Toth. A parallel shortest path algorithm for the assignment problem. Technical Report Management Science Report MSRR 552, Carnegie Mellon University, April 1989.

[5] Imre Bárány and Tibor Fiala. Többgépes ütemezési problémák közel optimális megoldása. *Szigma-Mat.-Közgazdasági Folyóirat*, 15:177–191, 1982.

[6] I.S. Belov and Ya. N. Stolin. An algorithm in a single path operations scheduling problem. In *Mathematical Economics and Functional Analysis [In Russian]*, pages 248–257. Nauka, Moscow, 1974.

[7] S. Ben-David, A. Borodin, R.M. Karp, G. Tardos, and A. Wigderson. On the power of randomization in on-line algorithms. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, pages 379–386, May 1990.

[8] B. Berger and L. Cowen. Complexity results and algorithms for $\{<, \leq, \}$-constrained scheduling. In *Proceedings of the 2nd ACM-SIAM Symposium on Discrete Algorithms*, pages 137–148, 1991.

[9] D. P. Bertsekas. A distributed algorithm for the assignment problem. Technical report, Laboratory for Information and Decision Sciences, MIT, 1979. Working Paper.

[10] D. P. Bertsekas. The auction algorithm: A distributed algorithm for the assignment and other network flow problems. *Annals of Operations Research*, 14:105–123, 1988.

[11] D. P. Bertsekas and D. Castanon. Parallel synchronous and asynchronous implementationsof the auction algorithm. Technical Report TP-308, Alphatech Inc., Burlington MA, November 1989.

[12] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice Hall, 1989.

[13] D.P. Bertsekas and J. Eckstein. Dual coordinate step methods for linear network flow problems. *Mathematical Programming*, 42:202–243, 1988.

[14] G. Blelloch. Scans as primitive parallel operations. In *Proceedings International Conference on Parallel Processing*, pages 355–362, August 1987.

[15] A. Blum, P. Raghavan, and B. Schieber. Navigating in unfamiliar geometric terrain. In *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*, pages 379–386, May 1991.

[16] L.D. Bodin, B.L. Golden, A.A. Assad, and M.O. Ball. Routing and scheduling of vehicles and crews. *Comp. and Oper. Res.*, 10:65–211, 1983.

[17] A. Borodin, N. Linial, and M. Saks. An optimal online algorithm for metrical task systems. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 379–386, May 1987.

[18] J.L. Bruno and P.J. Downey. Probabilistic bounds on the performance of list scheduling. *SIAM Journal on Computing*, 15:409–417, 1986.

[19] D. Castanon, B. Smith, and A. Wilson. Performance of parallel assignment algorithms on different multiprocessor architectures. Argonne National Lab. Report, in preparation, 1989.

[20] Y. Censor and S.A. Zenios. Massively parallel row-action algorithms for some nonlinear transportation problems. *SIAM J. Optimization*, 1:373–400, 1991.

[21] B. Chandra, H.J. Karloff, and S. Vishwanathan. Private communication, 1991.

[22] Y. Cho and S. Sahni. Bounds for list schedules on uniform processors. *SIAM Journal on Computing*, 9(1):91–103, February 1980.

[23] M. Chrobak, H. Karloff, T. Payne, and S. Vishwanathan. New results on server problems. In *Proceedings of the 1st ACM-SIAM Symposium on Discrete Algorithms*, pages 291–300, 1990.

[24] M. Chrobak and L.L. Larmore. An optimal on-line algorithm for k-servers on trees. *SIAM Journal on Computing*, 20(1):144–148, February 1991.

[25] I. Cidon, S. Kutten, Y. Mansour, and D. Peleg. Greedy packet scheduling. In *Proceedings of the 4th Annual Workshop on Distributed Algorithms, Bari, Italy*, 1990.

[26] Thinking Machines Corporation. Model CM-2 technical summary. Technical Report Report HA87-4, Thinking Machines Corporation, Cambridge, Massachusetts, April 1987.

[27] W.H. Cunningham and A.B. Marsh. A primal algorithm for optimum matching. *Mathematical Programming Study*, 8, 1978.

[28] E. Davis and J.M. Jaffe. Algorithms for scheduling tasks on unrelated processors. *Journal of the ACM*, 28:712–736, 1981.

[29] X. Deng and C.H. Papadimitriou. Exploring an unknown graph. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science*, October 1990.

[30] J. M. Draper. Compiling in horizon. In *Proceedings of Supercomputing '88*, November 1988.

[31] J. Eckstein. Implementing and running the alternating step method on the Connection Machine CM-2. *ORSA Journal of Computing.* to appear.

[32] J. Eckstein. *Splitting Methods for Monotone Operators with Applications to Parallel Optimization.* PhD thesis, Dept. of Civil Engineering, MIT, Cambridge, MA,. 1989.

[33] J. Eckstein and D.P. Bertsekas. On the Douglas-Rachford splitting method and the proximal point algorithm for maximal monotone operators. Technical Report CICS-P-167, Brown/Harvard/MIT Center for Intelligent Control Systems, 1989. To appear in Mathematical Programming.

[34] J. Eckstein and D.P. Bertsekas. An alternating direction method for linear programming. Technical Report Working Paper 90-063, Harvard Business School, 1990.

[35] Jr. E.G. Coffman and E.N. Gilbert. On the expected relative performance of list scheduling. *Operations Research*, 33:548–561, 1985.

[36] D.G. Feitelson and L.R. Rudolph. Mapping and scheduling in a shared parallel environment using distributed hierarchical control. In *Proceedings of the 1990 International Conference on Parallel Processing*, August 1990.

[37] D.G. Feitelson and L.R. Rudolph. Wasted resources in gang scheduling. In *Proceedings of the 5th Jerusalem Conference on Information Technology*, October 1990.

[38] A. Feldmann, J. Sgall, and S. Teng. Dynamic scheduling on parallel machines. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*, October 1991.

[39] Tibor Fiala. Közelítő algorithmus a három gép problémára. *Alkalmazott Matematikai Lapok*, 3:389–398, 1977.

[40] Tibor Fiala. An algorithm for the open-shop problem. *Mathematics of Operations Research*, 8(1):100–109, 1983.

[41] A. Fiat, Y. Rabani, and Y. Ravid. Competitive k-server algorithms. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science*, 1990.

[42] T. Fischer, A. V. Goldberg, and S. Plotkin. Approximating matchings in parallel. submitted for publication, July 1991.

[43] H. N. Gabow and R. E. Tarjan. Almost-optimum parallel speed-ups of algorithms for bipartite matching and related problems. Technical Report CU-CS-425-89, University of Colorado, Boulder, CO, January 1989.

[44] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, 1979.

[45] A. V. Goldberg. *Efficient graph algorithms for sequential and parallel computers.* PhD thesis, MIT, Cambridge, MA, January 1987.

[46] A. V. Goldberg, S. A. Plotkin, and P. M. Vaidya. Sublinear-time parallel algorithms for matching and related problems. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, pages 174–185, 1988.

[47] A. V. Goldberg and R. E. Tarjan. Solving minimum-cost flow problems by successive approximation. *Mathematics of Operations Research*, 15(3):430–466, 1990.

[48] A.V. Golderg, S. A. Plotkin, D.B. Shmoys, and E. Tardos. Interior point methods in parallel computation. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science.* IEEE Computer Society, October 1989.

[49] L. Goldschlager, R. Shaw, and J. Staples. The maximum flow problem is log-space complete for *P*. *Theoretical Computer Science*, 21:105–111, 1982.

[50] T. Gonzalez and S. Sahni. Open shop scheduling to minimize finish time. *Journal of the ACM*, 23:665–679, 1976.

[51] T. Gonzalez and S. Sahni. Flowshop and jobshop schedules: complexity and approximation. *Operations Research*, 26:36–52, 1978.

[52] R.L. Graham. Bounds for certain multiprocessor anomalies. *Bell System Technical Journal*, 45:1563–1581, 1966.

[53] R.L. Graham. Bounds on multiprocessing anomalies. *SIAM Journal of Applied Mathematics*, 17:263–269, 1969.

[54] E. Grove. The harmonic online k-server algorithm is competitive. In *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*, pages 260–266, May 1991.

[55] M. Habbal. Personal communication.

[56] L. Hall and D. B. Shmoys. Approximation schemes for constrained scheduling problems. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, pages 134–141. IEEE, October 1989.

[57] M.R. Hestenes. Multiplier and gradient methods. *JOTA*, 4:303–320, 1969.

[58] D. Hillis. *The Connection Machine.* MIT Press, 1985.

[59] D.S. Hochbaum and D.B. Shmoys. A best possible parallel approximation algorithm for a graph theoretic problem. In *Proceedings of IFORS '87*, pages 933–938, 1987.

[60] D.S. Hochbaum and D.B. Shmoys. Using dual approximation algorithms for scheduling problems: theoretical and practical results. *Journal of the ACM*, 34:144–162, 1987.

[61] D.S. Hochbaum and D.B. Shmoys. A polynomial approximation scheme for machine scheduling on uniform processors: using the dual approximation approach. *SIAM Journal on Computing*, 17:539–551, 1988.

[62] E.C. Horvath, S. Lam, and R. Sethi. A level algorithm for preemptive scheduling. *Journal of the ACM*, 24:32–43, 1977.

[63] S. Irani. Coloring inductive graphs on-line. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science*, pages 470–479, 1990.

[64] J.M. Jaffe. Efficient scheduling of tasks without full use of processor resources. *Theoretical Computer Science*, 12:1–17, 1980.

[65] D. S. Johnson. personal communication, 1991.

[66] S. M. Johnson. Optimal two- and three-stage production schedules with setup times included. *Naval Research Logistics Quarterly*, pages 61–68, 1954.

[67] R. Jonker and T. Volgenant. A shortest augmenting path algorithm for dense and sparse linear assignment problems. *Computing*, 38:325–340, 1987.

[68] Bala Kalyanasundaram and K. Pruhs. On-line weighted matching. In *Proceedings of the 2nd ACM-SIAM Symposium on Discrete Algorithms*, pages 234–241, 1991.

[69] Howard Karloff. A Las Vegas RNC algorithm for maximum matching. *Combinatorica*, 6(4):387–391, 1986.

[70] R. Karp, E. Upfal, and A. Wigderson. Constructing a perfect matching is in random NC. *Combinatorica*, 6:35–48, 1986.

[71] R. Karp, U.V. Vazirani, and V.V. Vazirani. On-line algorithms for bipartite matching. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, pages 352–358, 1990.

[72] R. Karp, U.V. Vazirani, and V.V. Vazirani. Online algorithms for transportation and matching problems. Unpublished manuscript, 1991.

[73] R. M. Karp and V. Ramachandran. A survey of parallel algorithms for shared-memory machines. In *Handbook of Theoretical Computer Science A*, pages 869–942. Elsevier, Amsterdam, 1990.

[74] D. Kempa, J. Kennington, and H. Zaki. Performance characteristics of the jacobi and gauss-seidel versions of the auction algorithm on the Alliant FX/8. Technical Report OR-89-008, Department of Mechanical and Industrial Engineering, University of Illinois, Champaign-Urbana, 1989.

[75] J. Kennington and Z. Wang. Solving dense assignment problems on a shared memory multiprocessor. Technical Report 88-OR-16, Dept. of Operations Research and Applied Science, Southern Methodist University, October 1988.

[76] L. Kirousis, M. Serna, and P. Spirakis. The parallel complexity of the subgraph connectivity problem. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, pages 294–299, 1989.

[77] L. Kirousis and P. Spirakis. Probabilistic log-space reductions and problems probabilistically hard for P. In *Proceedings of First Scandinavian Workshop on Algorithm Theory*, 1988.

[78] B.J. Lageweg, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinooy Kan. Computer-aided complexity classification of combinatorial problems. *Communications of the ACM*, pages 817–822, 1982.

[79] E.L. Lawler and J. Labetoulle. On preeemptive scheduling of of unrelated parallel processors by linear programming. *Journal of the ACM*, 25:612–619, 1978.

[80] E.L. Lawler, J.K. Lenstra, A.H.G. Rinooy Kan, and D.B. Shmoys, editors. *The Travelling Salesman Problem.* John Wiley and Sons, 1985.

[81] E.L. Lawler, J.K. Lenstra, A.H.G. Rinooy Kan, and D.B. Shmoys. Sequencing and scheduling: Algorithms and complexity. Technical Report BS-R8909, Centre for Mathematics and Computer Science, Amsterdam, The Netherlands, 1989. To appear in Handbooks in Operations Research and Management Science, Volume 4: Logistics of Production and Inventory.

[82] Tom Leighton, Bruce Maggs, and Satish Rao. Universal packet routing algorithms. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, pages 256–269, 1988.

[83] J.K. Lenstra, D.B. Shmoys, and É. Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical Programming*, 46:259–271, 1990.

[84] G. F. Lev, N. Pippenger, and L. G. Valiant. A fast parallel algorithm for routing in permutation networks. *IEEE Transactions on Computers*, C-30:93–100, 1981.

[85] L. Lovász. 2-matchings and 2-covers of hypergraphs. *Acta Math. Acad. Sci. Hungar.*, 26:433–444, 1975.

[86] L. Lovasz and M.Plummer. *Matching Theory.* North Holland, Amsterdam, 1986.

[87] M.S. Manasse, L.A. McGeoch, and D.D. Sleator. Competitive algorithms for on-line problems. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, pages 322–333, May 1988.

[88] Y. Mansour and B. Patt-Shamir. Greedy packet scheduling on shortest paths. In *Proceedings of the 1991 ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 1991. to appear.

[89] R. McNaughton. Scheduling with deadlines and loss functions. *Management Science*, 6:1–12, 1959.

[90] K. Mulmuley, U.V. Vazirani, and V.V. Vazirani. Matching is as easy as matrix inversion. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 345–354, 1987.

[91] P.R. Narayanan. personal communication, 1991.

[92] J.B. Orlin. personal communication, 1991.

[93] C. Papadimitriou and M. Yannakakis. Towards an architecture-independent analysis of parallel algorithms. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, pages 510–514, 1988.

[94] C. Papadimitriou and M. Yannakakis. Shortest paths without a map. In *Proceedings of the 1989 ICALP Conference*, pages 610–620, 1989.

[95] C. Phillips and S. A. Zenios. Experiences with large scale network optimization on the Connection Machine. In *Impact of Recent Computer Advances on Operations Research*. Elsevier Science Publishing Co., New York, NY, 1989.

[96] C. A. Phillips. *Theoretical and Experimental Analyses of Parallel Combinatorial Algorithms*. PhD thesis, MIT, Cambridge, MA, October 1989.

[97] S. Plotkin, D. B. Shmoys, and E. Tardos. Fast approximation algorithms for fractional packing and covering problems. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*, 1991. To appear.

[98] M.J.D. Powell. A method for nonlinear constraints in minimization problems. In *Optimization*. Academic Press, 1969.

[99] P. Raghavan. Probabilistic construction of deterministic algorithms: approximating packing integer programs. *Journal of Computer and System Sciences*, 37:130–143, 1988.

[100] P. Raghavan and M. Snir. Memory vs. randomization in on-line algorithms. In *Proceedings of the 1989 ICALP Conference*, 1989.

[101] P. Raghavan and C. D. Thompson. Provably good routing in graphs: regular arrays. In *Proceedings of the 17th Annual ACM Symposium on Theory of Computing*, pages 79–87, 1985.

[102] P. Raghavan and C. D. Thompson. Randomized rounding: a technique for provably good algorithms and algorithmic proofs. *Combinatorica*, 7:365 – 374, 1987.

[103] S. Sahni and Y. Cho. Nearly on line scheduling of a uniform processor system with release times. *SIAM Journal on Computing*, 8:275–285, 1979.

[104] A. Schrijver. Min-max results in combinatorial optimization. In A. Bachem, M. Grötschel, and B. Korte, editors, *Mathematical Programming, the state of the art: Bonn*, pages 439–500. sv, 1983.

[105] András Sebö. Finding the t-join structure of graphs. *Mathematical Programming*, 36:123–134, 1986.

[106] M. Serna. Approximating linear programming is log-space complete for $\mathcal{P}$. *Information Processing Letters*, 37:233–236, 1991.

[107] M. Serna and P. Spirakis. The approximability of problems complete for $\mathcal{P}$. In *Proceedings of the 2nd International Symposium on Optimal Algorithms*, pages 193–205, 1989. Published as Lecture Notes in Computer Science 401, Springer-Verlag.

[108] S. V. Sevast'yanov. On an asymptotic approach to some problems in scheduling theory. In *Abstracts of papers at 3rd All-Union Conf. of Problems of Theoretical Cybernetics [in Russian]*, pages 67–69. Inst. Mat. Sibirsk. Otdel. Akad. Nauk SSSR, Novosibirsk, 1974.

[109] S.V. Sevast'yanov. Efficient construction of schedules close to optimal for the cases of arbitrary and alternative routes of parts. *Soviet Math. Dokl.*, 29(3):447–450, 1984.

[110] S.V. Sevast'yanov. Bounding algorithm for the routing problem with arbitrary paths and alternative servers. *Kibernetika*, 22(6):74–79, 1986. Translation in Cybernetics 22, pages 773-780.

[111] P.D. Seymour. On odd cuts and plane multicommodity flows. *Proceedings of the London Mathematical Society*, 3(42):178–192, 1981.

[112] D. B. Shmoys, C. Stein, and J. Wein. Improved approximation algorithms for shop scheduling problems. In *Proceedings of the 2nd ACM-SIAM Symposium on Discrete Algorithms*, pages 148–157, January 1991.

[113] D. B. Shmoys, J. Wein, and D.P. Williamson. Scheduling parallel machines on-line. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*, October 1991. To appear.

[114] D.B. Shmoys. Personal communication, 1990.

[115] D.D. Sleator and R.E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[116] D.D. Sleator and R.E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32:652–686, 1985.

[117] C. Stein and J. Wein. Approximating the minimum-cost maximum flow is P-complete. in preparation, 1991.

[118] R. E. Tarjan. *Data structures and network algorithms.* SIAM, Philadelphia, PA, 1983.

[119] S. Vishwanathan. Randomized online coloring of graphs. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science*, pages 464–469, 1990.

[120] V. Le Phan Vu. Massively parallel solutions to the sparse assignment problem. Bachelor's thesis, MIT, 1991.

[121] J. Wein. Experience with minimum-cost circulation on the Connection Machine. Internal Thinking Machines Corporation Manuscript, 1989.

[122] J. Wein and S.A. Zenios. On the massively parallel solution of the assignment problem. *Journal of Parallel and Distributed Computing.* To appear.

[123] J. Wein and S.A. Zenios. Massively parallel auction algorithms for the assignment problem. In *Proceedings of the 3rd Symposium on the Frontiers of Massively Parallel Computation*, pages 90–99, 1990.

[124] D. P. Williamson. The non-approximability of shop scheduling. Unpublished Manuscript, 1991.

[125] H. Zaki. A comparison of two algorithms for the assignment problem. Report ORL 90-002, Dept. of Mechanical and Industrial Engineering, University of Illinois at Urbana-Champaign, 1990.

[126] S.A. Zenios. On the fine-grain decomposition of multicommodity transportation problems. Working paper, Decision Sciences Department, The Wharton School, University of Pennsylvania, 1990.

[127] S.A. Zenios and R.A. Lasken. Nonlinear network optimization on a massively parallel Connection Machine. *Annals of Operations Research*, 14:147–165, 1988.

[128] S.A. Zenios and S. Nielsen. Massively parallel row-action algorithms for nonlinear stochastic network programs. Working paper, Decision Sciences Department, The Wharton School, University of Pennsylvania, 1990.